



**Politecnico  
di Torino**

Master Degree in Cybersecurity

Master Degree Thesis

# **Towards a query-driven approach for the verification of Kubernetes configuration**

## **Supervisors**

prof. Fulvio Valenza  
prof. Daniele Brighenti  
prof. Riccardo Sisto  
dott. Francesco Pizzato

**Candidate**

Luca NOBILI

ACADEMIC YEAR 2024-2025



# Summary

Securing production Kubernetes clusters is a critical and increasingly complex task. The decentralized and fragmented nature of security policies, such as Role-Based Access Control (RBAC) and NetworkPolicy resources, creates a significant semantic gap between high level security requirements and the low level reality of the cluster configuration. Manual verification is cognitively demanding, error prone, and does not scale, while existing automated tools are often siloed within a single domain, leaving blind spots at the intersection of authorization and network reachability.

To bridge this gap, this thesis introduces a query-based verification approach that assesses cross-domain security properties, specifically across RBAC and NetworkPolicy, through a unified cluster model. This methodology is built upon three core components. First, a high-level, declarative query language has been established, engineered for modularity and extensibility to enable operators to formulate single-domain and cross-domain security questions. Second, a typed and unified graph model has been designed to normalize disparate Kubernetes configuration objects and policies into a unified and consistent representation of the cluster's state. Third, SWI-Prolog is employed as the formal engine to evaluate these high-level queries. To achieve this, we materialize the graph model as a SWI-Prolog knowledge base and compile the query language into a fixed set of Prolog rules. Operator questions are then posed as queries against this knowledge base and evaluated by the SWI-Prolog engine.

Although this work unifies the critical domains of RBAC and NetworkPolicy, the core architecture was designed for modularity, making it straightforward to incorporate additional Kubernetes policy and configuration domains. This modular design is validated by a Proof-of-Concept (PoC) that effectively answers both single- and cross-domain questions on realistic configurations, establishing a solid foundation for future integration.

# Contents

<b>List of Figures</b>	7
<b>List of Tables</b>	8
<b>Listings</b>	9
<b>1 Introduction</b>	11
1.1 Thesis Objective . . . . .	12
1.2 Thesis description . . . . .	13
<b>2 Background: Kubernetes</b>	15
2.1 Definition . . . . .	15
2.2 Kubernetes Cluster Architecture . . . . .	16
2.3 The Control Plane . . . . .	17
2.3.1 The Worker Node . . . . .	19
2.4 The Kubernetes Object Model and Core Resources . . . . .	20
2.4.1 Foundational Concepts: Labels, Selectors, and Namespaces . . . . .	21
2.4.2 Pod: The Atomic Unit of Scheduling . . . . .	22
2.4.3 Workload Management: Controllers and Abstractions . . . . .	22
2.4.4 Service Abstraction and Networking . . . . .	23
2.4.5 Configuration and Persistent State . . . . .	23
<b>3 Access Control and Network Policy in Kubernetes</b>	25
3.1 Access Control in Kubernetes: Overview . . . . .	25
3.2 A Deep Inspection of Role-Based Access Control (RBAC) . . . . .	28
3.2.1 Particular Cases in RBAC . . . . .	31
3.3 Network Policy: Data-Plane Segmentation . . . . .	34
3.3.1 From a Flat Network to Intentional Segments . . . . .	34
3.3.2 Anatomy of a <code>NetworkPolicy</code> . . . . .	34

<b>4</b>	<b>Thesis Objective</b>	<b>37</b>
4.1	A Taxonomy of High-Level Security Queries . . . . .	38
<b>5</b>	<b>Architecture Description</b>	<b>43</b>
5.1	Architecture Description . . . . .	44
5.2	Architectural Components in Detail . . . . .	45
5.2.1	Cluster Representation: Scope, Content, and Rationale . . .	45
5.2.2	Graph Representation of Cluster State and Policy . . . . .	46
5.2.3	High-Level Operator Queries . . . . .	51
5.2.4	Translator: From Unified Graph to Logic Program . . . . .	52
5.2.5	Result and Operator Presentation . . . . .	54
<b>6</b>	<b>Language Description</b>	<b>55</b>
6.1	RBAC Value Domains . . . . .	55
6.1.1	Resource . . . . .	56
6.1.2	Actions . . . . .	57
6.1.3	Permissions . . . . .	58
6.1.4	Subject . . . . .	58
6.1.5	Role . . . . .	59
6.2	NetworkPolicy Value Domains . . . . .	59
6.2.1	Protocols . . . . .	60
6.2.2	Ports . . . . .	60
6.2.3	Network Endpoints . . . . .	60
6.3	Predicate Queries . . . . .	61
6.3.1	Query Syntax and Semantics . . . . .	61
6.3.2	Examples . . . . .	63
6.3.3	Predicate Query Negation . . . . .	64
6.3.4	Combining Predicate Queries . . . . .	64
6.4	Function Queries . . . . .	65
6.4.1	Query Syntax and Semantics . . . . .	65
6.4.2	Examples . . . . .	68
6.4.3	Combining Function Queries . . . . .	69
6.4.4	Cardinality Queries as Function-Query Extensions . . . . .	71
6.5	Advanced Queries: Discovery and Type-Driven Auditing . . . . .	72
6.6	Example Table . . . . .	73

<b>7</b>	<b>Proof of Concept: Implementation and Validation</b>	<b>75</b>
7.1	Implementation . . . . .	76
7.1.1	Simplifications Adopted in the PoC and Their Impact on Validity . . . . .	76
7.1.2	Cluster Information Acquisition . . . . .	77
7.1.3	Unified Graph Construction . . . . .	78
7.1.4	Translating the Graph into SWI-Prolog . . . . .	80
7.1.5	The Analysis Logic: Prolog Rules and Queries . . . . .	82
7.2	Validation . . . . .	83
7.2.1	Cluster Description . . . . .	84
7.2.2	Query Examples and Results . . . . .	85
7.2.3	Evaluation Time and Bottlenecks . . . . .	93
<b>8</b>	<b>Conclusions</b>	<b>95</b>
8.1	Future Work . . . . .	96
	<b>Bibliography</b>	<b>98</b>

# List of Figures

2.1	Conceptual view of the Kubernetes cluster architecture . . . . .	17
3.1	Access control overview in Kubernetes: (1) authentication, (2) authorization, (3) admission control, (4) persistence to storage. . . . .	26
5.1	High-level data flow of the security query architecture. . . . .	44
5.2	Example of the unified, typed graph model showing structural containment, RBAC overlays (GRANT, BIND), and NetworkPolicy connectivity (CONNECT). . . . .	48

# List of Tables

2.1	Key controllers managed by the <code>kube-controller-manager</code> . . . . .	19
4.1	Operator-friendly query families with concise intent and examples. .	41
6.1	Valid Selector and Predicate combinations for a Subject root. . . . .	67
6.2	Valid Selector and Predicate combinations for a Role root. . . . .	67
6.3	Valid Selector and Predicate combinations for a Network Endpoint root. . . . .	68
6.4	Operator-friendly query families with concise DSL representation and examples. . . . .	73
7.1	Environment, tools, and versions used in the PoC . . . . .	76
7.3	Indicative PoC timings and sizes (single run, <code>webapp</code> snapshot) . . .	94
7.4	Indicative Prolog goal evaluation times (mean over warm runs; PoC snapshot) . . . . .	94

# Listings

2.1	Minimal, generic Kubernetes object manifest. . . . .	21
3.1	A Role granting read-only access to Pods and a single, named ConfigMap. . . . .	29
3.2	A ClusterRole for viewing nodes, all Pods, and scaling Deployments. . . . .	29
3.3	A RoleBinding granting a namespaced Role to multiple subjects. . . . .	30
3.4	A ClusterRoleBinding granting cluster-wide permissions to a group. . . . .	30
3.5	A RoleBinding scoping a ClusterRole's permissions to a single namespace. . . . .	31
3.6	Two forms of role aggregation: (A) contributing to a built-in role, and (B) a custom umbrella role. . . . .	32
3.7	Minimal, generic NetworkPolicy skeleton. . . . .	35



# Chapter 1

## Introduction

Over the past decade, software delivery has shifted decisively toward cloud-native architectures. Organizations increasingly deploy microservices packaged as containers and orchestrated at scale, seeking portability, elasticity, and rapid release cycles. Kubernetes (K8s) has emerged as the de facto standard for container orchestration, powering platforms that span public, private, and hybrid clouds. While this shift accelerates innovation, it also multiplies the number of control points and configuration artifacts that must be secured and continuously verified. In practice, security posture in Kubernetes hinges on *policies*: access control (RBAC), network segmentation (NetworkPolicies), admission and governance controls, and related configuration constraints [1, 2, 3].

Unfortunately, real-world experience and literature highlight how difficult it is to reason about these policies at scale. The OWASP *Kubernetes Top Ten* explicitly calls out overly permissive RBAC, missing network segmentation, and lack of centralized policy enforcement as systemic risks [4]. NSA and CISA similarly emphasize that Kubernetes' flexibility, distributed control plane, and large configuration surface create fertile ground for misconfiguration and privilege escalation, recommending defense-in-depth controls and rigorous auditing [1]. From an adversarial perspective, the MITRE ATT&CK framework for containers underscores network segmentation and strict account management as core mitigations against common tactics such as lateral movement and container abuse [5].

This landscape of technical vulnerabilities intersects with an increasingly stringent regulatory climate, making continuous auditing and compliance validation essential. Directives such as the NIS2 and regulations like the GDPR mandate that organizations deploy suitable technical and administrative controls to protect sensitive information and guarantee system durability. For platforms utilizing Kubernetes, this translates into the demonstrable monitoring and enforcement of security policies controlling network communication, data accessibility, and the privileges assigned to workloads. Failing to supply this necessary assurance not only generates security holes ripe for exploitation but also exposes the organization to considerable legal, financial, and reputational risk.

Empirical studies and emerging research confirm these concerns at a technical level. Large-scale analyses of open-source Kubernetes manifests report widespread security misconfigurations that are hard to detect during code review alone [6]. In

response, researchers are exploring formal methods encoding RBAC and admission policies as logical constraints to detect unintended permissions before deployment[7]. Similar efforts target network isolation, proposing automated validation of `NetworkPolicy` objects to close segmentation gaps that are otherwise difficult to spot across namespaces and CNI implementations [1, 3]. This growing body of scientific literature highlights a clear need for tooling that moves beyond manual checks toward automated, intent-based reasoning.

## 1.1 Thesis Objective

Securing a production Kubernetes cluster requires operators to reason about a complex, distributed, and dynamic set of policies. To assess risk, they must manually correlate RBAC rules, bindings, and `NetworkPolicy` objects that are often scattered across numerous namespaces and subject to constant change. This manual approach is cognitively demanding, slow, and highly susceptible to error, making it trivial to overlook a subtle interaction between rules or for minor configuration drift to introduce an unintended permission or an insecure network path. Prior work in policy management and firewall analysis confirms that manual reviews do not scale and tend to accumulate anomalies over time; without automated tooling to normalize and explain the cumulative effect of policies, security posture inevitably degrades [8, 9, 10]. This creates a significant semantic gap between an administrator’s high-level security intent and the low-level, fragmented reality of the cluster’s configuration.

This thesis introduces an **intent-based query language specifically designed for verifying Kubernetes security policies**. The primary goal is to enable system administrators to pose complex security queries using high-level, natural, and operator-focused terminology. In return, the system provides a deterministic, verifiable result accompanied by a concise rationale. For example, an operator could formally verify assertions such as, “*Only ServiceAccounts within the CI grouping are authorized to modify Deployments located in the `staging` namespace,*” or “*No Pod residing in the `payments` namespace is network-accessible from external sources, except for traffic on TCP port 443.*” The foundational design of this language rests on three pillars: it must be (i) **direct**, using the vocabulary familiar to operators rather than the complex syntax of a formal verification solver; (ii) **intuitive**, built upon the core concepts of who (subjects), what (actions), where (resources), and how (network flows); and (iii) **extensible**, ensuring future compatibility with new policy domains without requiring major architectural overhaul.

This approach is grounded in a significant body of research that supports the shift from ad-hoc configuration checks to machine-checked reasoning based on formal intent. In access control, established methodologies allow for the refinement of policies from high-level goals and their subsequent verification, thereby reducing human error [11, 12]. Similarly, in networking, formal models have proven effective for verifying that network reachability complies with high-level constraints, even in complex virtualized environments [13, 14]. Within the Kubernetes ecosystem itself, recent work on intent-based security automation has demonstrated that high-level

specifications shorten audit cycles and reduce configuration mistakes [15, 16]. Synthesizing these findings, this thesis argues that an operator-facing, intent-driven query language is a practical and effective means to reduce review times, mitigate the risk of human error, and generate explainable evidence for security audits and compliance.

To provide a focused and concrete validation, this work targets two of the most critical security domains in Kubernetes: RBAC (authorization) and `NetworkPolicy` (network segmentation). The central contribution of this thesis is the design and validation of a **unified query model** where an operator can ask a single question that spans both domains and receive a coherent answer. This unified view is critical because many significant risks emerge precisely at the intersection of authorization and reachability for example, a principal who holds an unexpected permission and can reach a sensitive API endpoint. While the implementation focuses on these two domains, the underlying model is designed for modular extension, preparing the ground for future adapters to incorporate other policy types, such as admission controls or Pod Security Standards, without altering the operator’s query experience[12, 9].

## 1.2 Thesis description

This thesis is organized to follow a logical progression from conceptual foundations to implementation and validation.

- **Chapter 2 & Chapter 3: Background** establishes the necessary technical foundation by detailing the **Kubernetes security model**, including the cluster architecture, core objects, and the principles of access control and network segmentation that underpin the system.
- **Chapter 4: Thesis Objective** defines the goals and research questions driving this work, clearly articulating the motivation for developing a unified, formal approach to policy verification.
- **Chapter 5: Architecture Description** presents the central architectural contribution, detailing how policy knowledge from multiple Kubernetes domains is integrated into a **typed graph model** and translated into a reasoning core for logic-based verification.
- **Chapter 6: Language Description** specifies the custom query language designed to interact with the model, formalizing its syntax, semantics, and the set of operators supported for expressing complex security invariants.
- **Chapter 7: Proof of Concept (PoC)** operationalizes the proposed framework through a working implementation of the graph compiler and reasoning pipeline, validating its effectiveness on representative **intra and inter-domain scenarios**.

- **Chapter 8: Conclusions and Future Work** summarizes the key findings, reflects on the limitations of the current implementation, and outlines directions for future extensions, such as the inclusion of new policy families and deployment in operational clusters.

# Chapter 2

## Background: Kubernetes

This chapter first delineates the architecture of a Kubernetes cluster: its control plane and worker nodes, clarifying the responsibilities and interactions of the principal components; it then formalizes the object model and core resources (labels/selectors, namespaces, Pods, and workload controllers) before addressing the networking and exposure layer (Services and Ingress) and, finally, configuration and state management (ConfigMaps, Secrets, and storage via PersistentVolumes/PersistentVolumeClaims/StorageClass). These sections establish the technical vocabulary and operational assumptions employed in the remainder of the thesis, particularly for the treatment of security policies and their verification.

### 2.1 Definition

Kubernetes (K8s) is an open-source platform for orchestrating containerized applications, presenting a cluster of potentially heterogeneous machines as a single, programmable system. Its operational paradigm is fundamentally declarative rather than imperative. Instead of executing a sequence of commands, users define a *desired state* for their applications and associated services. A distributed control plane then continuously observes the system's actual state and applies corrective actions to ensure it converges with the desired state. This model, which combines declarative intent with continuous reconciliation, transforms the underlying infrastructure into an automated substrate. Consequently, critical operations such as workload placement, health monitoring, scaling, and updates are governed by policy instead of being managed through ad-hoc scripts.

The widespread adoption of Kubernetes can be attributed to its ability to provide a consistent abstraction layer across diverse computing environments, including public clouds, private data centers, and hybrid or multi-cloud deployments. By decoupling application concerns from the specifics of the underlying infrastructure, Kubernetes affords significant portability and establishes a common operational language for technical teams. This allows platform engineers to standardize recommended workflows and configurations while still granting development teams the freedom to integrate specialized components for networking, storage, policy enforcement, and observability. The result is a versatile, shared foundation capable of

supporting a wide range of architectures, from large-scale microservice deployments to data processing pipelines, without committing an organization to a single vendor or tool-chain.

At a high level, Kubernetes provides several classes of capability that production operations demand. First, it automates placement and capacity management: workloads are scheduled onto available resources with respect to constraints and goals, improving utilization while honoring availability and locality. Second, it builds resilience into the runtime: when failures occur, the system detects them, replaces or relocates work, and gates traffic until components are healthy, thereby reducing mean time to repair. Third, it manages change safely: updates can be introduced progressively and evaluated against health signals so that issues are contained and reversals are orderly. Fourth, it separates configuration from compute artifacts and supports secure handling of sensitive runtime data, enabling the same binaries to move cleanly across environments. Fifth, it offers native pathways for elasticity and workload diversity, so always-on services, batch activities, and scheduled tasks can all be expressed and governed within one operational model. Finally, Kubernetes is designed for extensibility: its APIs and controllers can be augmented by higher-level automation and domain-specific abstractions, allowing organizations to evolve the platform without forking it. [17].

## 2.2 Kubernetes Cluster Architecture

The architecture of a Kubernetes cluster is that of a distributed system, comprising a collection of machines known as *nodes*. These nodes are categorized into two primary roles: the *control plane* and the *worker nodes*. The control plane functions as the orchestration and management core of the cluster. It serves as the authoritative source for the cluster's desired state and coordinates the actions required to make the actual state converge with that specification. Conversely, the worker nodes provide the execution environment where containerized workloads are run.

This core separation of responsibilities is fundamental to Kubernetes' scalability, resilience, and its declarative operational model. Users specify a desired state via the Application Programming Interface (API), and a set of autonomous control loops then operate continually to align the system's actual (observed) state with the user's declared objective. Figure 2.1 presents a conceptual summary of this architecture, detailing the essential components and how they interact.

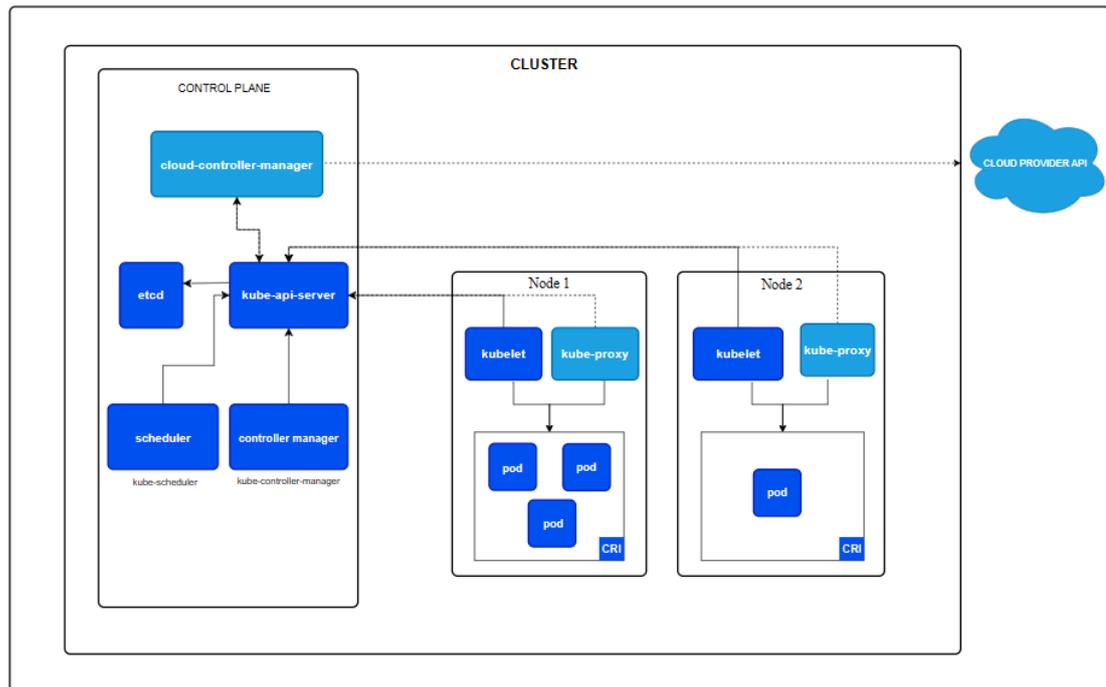


Figure 2.1. Conceptual view of the Kubernetes cluster architecture

## 2.3 The Control Plane

The control plane can be conceptualized as the central nervous system of a Kubernetes cluster. It is responsible for maintaining the system's desired state, making global scheduling decisions, and responding to cluster events. To ensure high availability and scalability, the components of the control plane are typically replicated across multiple dedicated *control plane nodes*, relying on mechanisms like leader election to coordinate actions.

**kube-apiserver** As the primary management gateway to the cluster, the API server exposes a RESTful Kubernetes API. It is the focal point for all administrative tasks, handling authentication, authorization, and a sequence of admission controllers that validate and potentially modify requests before persisting an object's state. A key architectural feature is its implementation of an efficient *watch* mechanism, which allows clients to subscribe to streams of changes. This enables an event-driven architecture, eliminating the need for inefficient polling. The API server is the *only* component that communicates directly with the `etcd` data store; all other components interact with the cluster's state exclusively through the API. Architecturally, it is designed to be stateless and horizontally scalable, with request-handling mechanisms to ensure fairness and prioritize critical system traffic under heavy load.

**etcd** `etcd` is a distributed key-value store engineered for strong consistency and high availability. It functions as the definitive persistence layer for all cluster configuration data and object states. By providing linearizable read and write operations, **etcd** ensures that other control plane components, such as the scheduler and controllers, can make reliable decisions based on a consistent view of the system, even during periods of high activity.

**kube-scheduler** The scheduler's main responsibility is assigning newly created **Pods** to an appropriate worker node. This decision involves two primary stages. First, a **filtering** process removes any nodes that fail to satisfy the Pod's mandatory constraints, which include things like resource requirements, node affinities, or tolerance for specific taints. Following this, a **scoring** stage ranks the remaining suitable nodes based on secondary preferences, such as distributing workloads across separate failure domains or placing related Pods together. The scheduler is also capable of preempting lower-priority workloads if required to accommodate a high-priority Pod. Its operations are extensible via a *scheduling framework* that facilitates the integration of custom decision-making logic.

**kube-controller-manager** This component is a daemon that packages the essential reconciliation loops, referred to as **controllers**, into one binary. Each controller functions as a dedicated process that monitors a specific API object type for changes and strives to bring the system's actual (observed) state into alignment with the state declared in the object's definition. These controllers typically operate autonomously and rely on a leader election mechanism to guarantee that only one instance is actively running at any moment, thereby ensuring the high availability of the reconciliation processes.

**cloud-controller-manager** The cloud controller manager acts as an intermediary layer, separating specialized cloud-provider logic from the core Kubernetes codebase. It interfaces with the native APIs of a cloud provider to manage essential resources such as the *node lifecycle* (e.g., detecting and configuring new VMs), *routing* (setting up network routes), and *load balancers* (provisioning and managing external load balancers for Services). This deliberate isolation preserves Kubernetes' ability to function across different cloud environments.

Table 2.1 presents a selection of the core controllers bundled within the **kube-controller-manager** and summarizes their primary responsibilities.

Table 2.1. Key controllers managed by the `kube-controller-manager`.

Controller	Function
Node Controller	Tracks the operational status and lifecycle of nodes, initiating reactions (like marking a node unhealthy) when heartbeats cease or conditions are reported.
Job Controller	Manages the complete execution of one time tasks ( <b>Jobs</b> ), provisioning Pods until the task reaches a successful conclusion, including managing failure based retries.
EndpointSlice Controller	Creates and ensures the integrity of <code>EndpointSlice</code> objects, which are used to track the specific network endpoints (such as Pod IPs) that back a Service.
Replication / ReplicaSet Controllers	Guarantees that the exact, specified number of Pod replicas for a workload is continuously maintained, either by scaling up or scaling down the Pod count.
Garbage Collector & Namespace Controllers	Executes essential maintenance duties, including the deletion of subordinate objects when their owner is removed (cascading deletion) and handling the finalization of namespaces.

### 2.3.1 The Worker Node

Worker nodes constitute the computational capacity of a Kubernetes cluster, providing the CPU, memory, networking, and storage resources required to execute containerized workloads. Each worker is registered within the cluster as a `Node` object in the API server and reports its health and status through periodic heartbeats. The node's state is treated as ephemeral; they are designed to be replaceable resources, and if a node fails, its workloads can be rescheduled elsewhere in the cluster to maintain the desired state. Placement of workloads onto specific nodes can be influenced through mechanisms such as labels, taints, and affinity rules.

**kubelet** The `kubelet` is the primary agent that runs on each worker node. Its core responsibility is to communicate with the control plane's API server and ensure that the containers described in Pod specifications (`PodSpec`) are running and healthy.

It translates these specifications into concrete actions for the container runtime, manages the entire Pod lifecycle (including liveness, readiness, and startup probes), and reports the status of the node and its resident Pods back to the control plane. The `kubelet` also serves as an integration point, interacting with plugins that adhere to the Container Runtime Interface (CRI), Container Network Interface (CNI), and Container Storage Interface (CSI) to manage containers, networking, and storage, respectively.

**kube-proxy** The `kube-proxy` is a network proxy that runs on each node and is a fundamental component of the Kubernetes networking model. Its primary function is to implement the Kubernetes `Service` concept. It does this by monitoring the API server for changes to `Service` and `EndpointSlice` objects and programming the node's underlying OS packet-handling subsystem (such as iptables or IPVS) to direct traffic to the correct backend Pods. This provides a stable virtual IP for in-cluster load balancing, decoupling service communication from the lifecycle of individual Pods. While alternative data planes using technologies like eBPF can replace `kube-proxy`, they still implement the same fundamental `Service` abstraction.

**Container Runtime** The `kubelet` delegates the direct management of containers to a container runtime via the Container Runtime Interface (CRI). This interface allows Kubernetes to be compatible with any compliant runtime, such as `containerd` or `CRI-O`. The runtime is responsible for tasks like pulling container images from a registry, creating and managing the container's execution environment (sandbox), and reporting low-level status. This strategic decoupling ensures that the Kubernetes project remains independent of any specific container implementation, fostering a diverse ecosystem of runtime technologies.

Cluster communications follow a hub-and-spoke model with the API server as the central chokepoint. Nodes initiate outbound, TLS-protected connections to the API server for watches, heartbeats, and status updates; authentication and authorization are enforced at the API layer. A smaller, explicitly enabled set of reverse flows from the API server to the `kubelet` support interactive operations such as `exec`, `attach`, `port-forward`, and `logs`. Hardening guidance focuses on mutual TLS for component traffic, disabling unauthenticated `kubelet` endpoints (e.g., ensuring the read-only port is off), minimizing node-exposed ports, and centralizing policy and audit at the API server to keep trust boundaries clear on untrusted networks. [18, 19]

## 2.4 The Kubernetes Object Model and Core Resources

Kubernetes models all user interactable entities as persistent API *objects*. Users declare the *desired state* of these objects typically by submitting YAML or JSON

manifests to the cluster and a set of controllers drives the system toward that state. Each object is identified by a *Kind* (the canonical type, such as `Pod` or `Service`) and is served under a specific API *group/version* (for example, `core/v1` or `apps/v1`). Objects are accessed through REST *resources*, i.e., collection endpoints like `/api/v1/pods` or `/apis/apps/v1/deployments`. Every object carries a human-chosen *name* that is unique within its scope (either a namespace or the whole cluster) and a server-assigned immutable *UID* that provides stable identity across renames or rescheduling. In addition to top-level fields, many objects expose *subresources* such as `/status` and `/scale` that partition responsibilities and authorization: clients may be authorized to read or update status without being able to modify the specification, or a controller may adjust scale independently of other fields. [20, 17]

A core semantic distinction runs through most Kubernetes objects: `spec` expresses the desired configuration, whereas `status` reports observed state and conditions. Controllers, agents, and the scheduler watch objects via the API's watch semantics and take incremental, idempotent actions until `status` converges toward `spec`. Lifecycle and ownership are captured through *owner references* and *finalizers*: owner references allow garbage collection of dependents when an owner is deleted, and finalizers defer deletion until clean-up work is complete.

Listing 2.1. Minimal, generic Kubernetes object manifest.

```

apiVersion: <GROUP>/<VERSION> # e.g., "v1" (core) or "apps/v1"
kind: <KIND> # e.g., "Pod", "Service", "Deployment"
metadata:
  name: <NAME> # unique within scope
  namespace: <NAMESPACE> # omit for cluster-scoped kinds
  labels: # selection/grouping
    <KEY>: <VALUE>
spec: # desired state (kind-specific)
  # ...
status: # observed state (set by the system)
  # ...

```

### 2.4.1 Foundational Concepts: Labels, Selectors, and Namespaces

**Labels and Selectors.** Labels are key–value pairs attached to objects for identification, grouping, and selection. They are intended for indexing and are therefore small, structured, and frequently queried. *Label selectors* provide the query language used throughout Kubernetes to relate objects: equality-based (`=`, `==`, `!=`) and set-based (`in`, `notin`, `exists`, `doesnotexist`) expressions allow clients and controllers to resolve a dynamic set of targets. This mechanism is pervasive: a `Service` selects the `Pods` that implement it; a `Deployment` (via its `ReplicaSet`) selects the `Pods` it manages; a `NetworkPolicy` selects the `Pods` to which its rules apply. Labels are distinct from *annotations*, which carry arbitrary, non-identifying metadata for tools and automation and are not used in selection. Some APIs also

support *field selectors*, which filter on a restricted set of object fields (for example, `metadata.name` or a Pod's `status.phase`). [21]

**Namespaces.** Namespaces partition a single physical cluster into multiple virtual clusters. They scope names, policies, and defaults; many resources (such as Pods and Services) are namespaced, whereas others (such as Nodes and PersistentVolumes) are cluster-scoped. Namespaces enable multi-tenancy and environment separation (e.g., development, staging, production), and they underpin quota and policy boundaries. Kubernetes ships with well-known namespaces such as `default`, `kube-system`, and `kube-public`, each used for specific categories of objects and components. [22]

## 2.4.2 Pod: The Atomic Unit of Scheduling

A **Pod** is the smallest deployable unit in Kubernetes and represents a single instance of an application workload. A Pod may contain one or more tightly coupled containers that are always scheduled together onto the same node. Containers in a Pod share the Pod's *network namespace* (they see the same IP and can communicate over `localhost`) and can mount the same volumes; optional settings allow sharing of additional namespaces (such as the process namespace) when appropriate. This co-location enables composition patterns commonly referred to as “sidecars,” where helper containers (for example, a proxy, a log shipper, or an init task that prepares data) run alongside the main application container. A Pod's lifecycle is described with probes and hooks: *liveness* probes determine when a container should be restarted, *readiness* probes gate traffic until the container is ready to serve, and *startup* probes accommodate long initialization times; lifecycle hooks provide callbacks at well-defined phases. Pod-level policies and resource settings (such as resource requests/limits, security context, and restart policy) inform placement and manageability. [23]

## 2.4.3 Workload Management: Controllers and Abstractions

Users rarely create and manage individual Pods directly. Instead, they express intent using *workload controllers* that maintain a stable population of Pods, replace failed instances, and coordinate changes over time.

**Deployments.** A **Deployment** manages stateless applications by creating and overseeing **ReplicaSets**, which in turn maintain the requested number of Pod replicas that match a label selector. Deployments support declarative updates and rollbacks by creating new ReplicaSets for revised Pod templates and gradually shifting the replica count from the old to the new set under configurable strategies. Scaling (horizontal) adjusts the number of replicas, while the selector binds the Deployment to the Pods it owns and is immutable to preserve consistency. [24]

**StatefulSets.** A `StatefulSet` manages stateful applications requiring stable network identities and persistent storage. It assigns an ordinal and a stable name to each Pod (e.g., `db-0`, `db-1`), supports ordered and graceful deployment, scaling, and deletion, and is commonly paired with per-Pod persistent volumes. Unlike stateless workloads, replacement Pods retain their identity, aiding stateful protocols and data partitioning. [25]

**DaemonSets.** A `DaemonSet` ensures that a copy of a Pod runs on all, or on a designated subset of, Nodes. This is used for node-level services—such as log collectors, monitoring agents, or networking components—that must be present everywhere or on specific classes of machines. When nodes are added to the cluster, the `DaemonSet` automatically schedules the daemon Pod onto them; when nodes are removed, the Pods are cleaned up. [26]

**Jobs and CronJobs.** A `Job` creates one or more Pods to run a task to completion, with configurable retry behavior and parallelism. A `CronJob` runs Jobs on a time-based schedule expressed with standard cron notation, enabling periodic or batch workflows. These abstractions are suited to finite workloads such as report generation, ETL tasks, and data maintenance. [27, 28]

#### 2.4.4 Service Abstraction and Networking

Once workloads are running, Kubernetes provides stable, discoverable endpoints for communication and, where needed, access from outside the cluster.

**Services.** A `Service` presents a stable virtual identity (a `ClusterIP` and a DNS name) for a logical set of Pods and distributes traffic across healthy backends matched by a label selector. Clients connect to the `Service`, not to individual Pod IPs, thereby decoupling callers from Pod churn due to rescheduling or scaling. `Service types` describe reachability: `ClusterIP` exposes an in-cluster virtual IP; `NodePort` allocates a port on each node to forward external traffic to the `Service`; `LoadBalancer` integrates with an external load balancer provided by the underlying infrastructure. Headless `Services` (`clusterIP: None`) expose individual Pod endpoints directly, which is useful for certain stateful or discovery scenarios. [29]

#### 2.4.5 Configuration and Persistent State

**ConfigMaps and Secrets.** A `ConfigMap` holds non-sensitive configuration data, and a `Secret` holds sensitive values such as credentials and tokens. Both decouple configuration from container images and can be projected into Pods as environment variables or mounted files. Secrets are transmitted over TLS between API clients and the API server and are stored base64-encoded by default; clusters can enable encryption at rest to protect `Secret` data in the backing store. [30, 31]

**PersistentVolumes (PVs), PersistentVolumeClaims (PVCs), and StorageClass.** A **PersistentVolume** represents a piece of provisioned storage in the cluster, while a **PersistentVolumeClaim** is a user's request for storage with specified size, access modes, and other characteristics. The control plane *binds* a claim to a suitable volume; when *dynamic provisioning* is enabled via a **StorageClass**, the cluster can create new volumes on demand through a storage driver (typically via the Container Storage Interface). Storage semantics are expressed with *access modes* (such as `ReadWriteOnce`, `ReadOnlyMany`, `ReadWriteMany`, and `ReadWriteOncePod`) and *volume modes* (`Filesystem` or `Block`), and lifecycle policy is governed by the volume's *reclaim policy* (for example, `Retain`, `Recycle` [deprecated], or `Delete`). This two-step model separates storage provisioning from consumption and promotes portability across diverse backends. [32]

# Chapter 3

## Access Control and Network Policy in Kubernetes

This chapter establishes the foundational mechanisms for security enforcement in Kubernetes, focusing on both the API control plane and the Pod data plane. The first half provides an operational overview of **Access Control in Kubernetes and RBAC**, detailing the end-to-end API request pipeline (authentication, authorization, and admission control) and conducting a deep inspection of the Role-Based Access Control (RBAC) object model. The second half introduces **Network Policy** as the critical Layer 3/4 enforcement tool for micro-segmentation, explaining its additive execution model, default-deny semantics, and object structure for governing Pod-to-Pod and external communication. This integrated framing covers the complete security posture required for cluster operations and workload isolation.

### 3.1 Access Control in Kubernetes: Overview

Kubernetes treats every interaction with cluster state as an HTTP request to the API server. Controlling access therefore means governing *who* can talk to the API, *what* they can ask it to do, and *under which constraints* requests are admitted and persisted. Figure 3.1 summarizes this end-to-end path, highlighting the numbered stages. In brief, each request traverses a pipeline: transport security (TLS, by default) → authentication (1) → authorization (2) → admission control (3) → validation and persistence (4) [33].

**Authentication** Every request to the Kubernetes API server first arrives over a TLS-secured connection; once the channel is established, the server attempts to *authenticate* the caller. Multiple *authenticator modules* can be enabled concurrently and are tried in sequence until one succeeds, common options include client certificates, bearer tokens (static or bootstrap), service-account JSON Web Tokens, and OIDC tokens issued by an external identity provider. A successful authenticator yields a *username* and, optionally, a set of *groups*; if no authenticator accepts the credentials, the request is rejected as unauthorized. Crucially, Kubernetes itself does not maintain a user directory: there is no `User` object in the API, and the

control plane learns a caller’s identity solely from the configured authenticators. [33, 34]

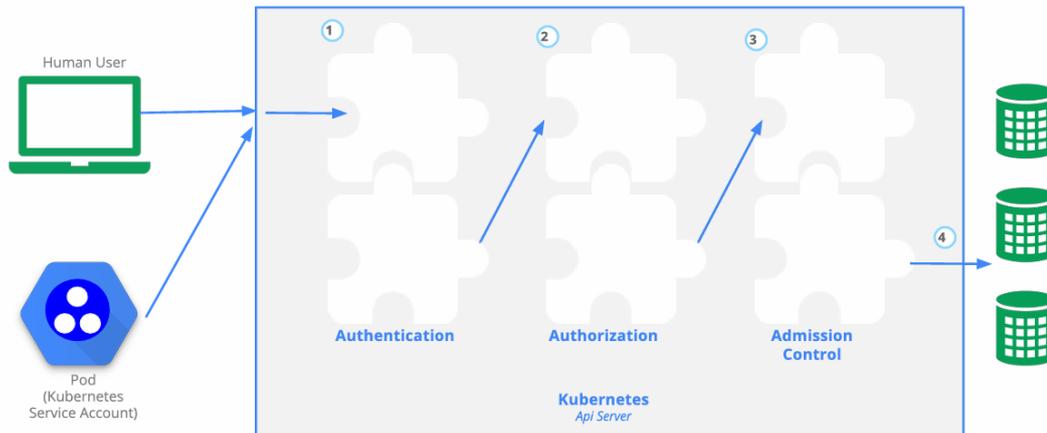


Figure 3.1. Access control overview in Kubernetes: (1) authentication, (2) authorization, (3) admission control, (4) persistence to storage.

**Human users vs. ServiceAccounts.** Kubernetes distinguishes human users from workloads. Human users are *external* to the cluster: their lifecycle (creation, revocation, group membership) is managed in an enterprise IdP or PKI, and the API server treats whatever identity the authenticator presents as authoritative. Inside the cluster, authorization policies (RBAC) therefore match users and groups by *string identity* exactly as asserted by the authenticator (for example, OIDC claims or certificate subjects). By contrast, a **ServiceAccount** is a first-class, namespaced API object that represents *workload identity*. Each namespace includes a `default` service account; Pods use it unless a different account is specified via `spec.serviceAccountName`. When a Pod runs with a service account, the kubelet typically mounts a *projected* token obtained through the `TokenRequest` API. Modern service-account tokens are short-lived, audience-bound (valid only for the intended server), and revocable, replacing older long-lived secret-backed tokens. [35, 34]

**Groups.** Groups are not Kubernetes objects; they are opaque strings asserted by the active authenticator (for example, OIDC group claims or certificate organization fields). In summary, authentication establishes a caller’s identity (user and groups, or service-account identity) that subsequent stages, authorization and admission, use to decide whether and how the request may change cluster state. [33]

**Authorization** Once Kubernetes authenticates a request, it must decide if that user or service is actually allowed to perform the requested action. This is handled

by one or more **authorizers**. Think of it like a series of security checkpoints: if **any single authorizer approves** the request, it is allowed to proceed. If all of them deny it, the server responds with a “Forbidden” (HTTP 403) error. An authorizer evaluates the attributes of an incoming request, including the identity of the principal (the user, group, or service account), the requested action (such as `create` or `delete`), and the target resource (for instance, a Pod within the `production` namespace). Kubernetes supports several authorization modes; the principal ones are summarized below.

**Role-Based Access Control (RBAC)** This is the current and preferred authorization mechanism for most modern Kubernetes installations. It offers a declarative and fully auditable system for managing access rights. The model revolves around defining sets of permissions called `Roles` or `ClusterRoles`, and subsequently linking these roles to specific users or service accounts using `RoleBindings` or `ClusterRoleBindings`.

**Node Authorizer** A specialized, internal authorizer designed to enforce the **least privilege** principle for the `kubelet` agents running on worker nodes. It ensures that a `kubelet` can only interact with or modify the API objects (like Pods) that are scheduled to or running on its host node. This significantly improves cluster security by minimizing the blast radius of a potential node compromise.

**Webhook Authorizer** This authorization method serves as an extension mechanism, outsourcing access control decisions to an external, administrator-specified service. Upon receiving a request, the Kubernetes API server forwards the request details to this remote webhook. The external service then evaluates the request against its own rules and returns a definitive “allow” or “deny” response, facilitating seamless integration with custom or third-party policy engines.

**Attribute-Based Access Control (ABAC)** This is an older, legacy authorization system that has been largely replaced by RBAC. ABAC grants permissions based on matching attributes within the request, using a static policy file stored on the API server. Due to its reduced flexibility and increased management complexity compared to RBAC, it is seldom utilized in production environments today.

**AlwaysAllow / AlwaysDeny** These basic authorizers are primarily intended for non-production use, such as testing or rapid prototyping. The `AlwaysDeny` setting blocks every incoming request, whereas `AlwaysAllow` grants permission to all requests. Extreme care is required with the `AlwaysAllow` authorizer; since authorization succeeds if *any* authorizer in the chain grants access, enabling it risks completely overriding and neutralizing all other active security policies.

**Admission Control** While authorization answers the question of *who may attempt what*, admission control provides the final judgment on *what is allowed to exist* in the cluster. After a request is successfully authenticated and authorized,

it enters the admission control phase, where it is processed sequentially by one or more configured **admission controllers**. These controllers serve as the ultimate policy enforcement mechanism before an object is persisted to storage. Admission controllers are categorized into two distinct types, which operate in order:

- **Mutating Controllers:** These controllers can modify incoming API objects. They are commonly used to enforce cluster-wide conventions by applying default values, injecting required fields or sidecar containers, or setting standardized labels and annotations.
- **Validating Controllers:** After all mutations are complete, validating controllers inspect the final object against a set of policies. They can either approve or reject the request based on its content, but they cannot modify it. This is the primary mechanism for enforcing security constraints, such as preventing the creation of privileged Pods or ensuring compliance with custom resource schemas.

Crucially, unlike the OR-composed logic of authorization, the admission control chain is AND-composed. A request must be approved by *every* admission controller in the sequence. If any single controller rejects the request, the entire operation fails immediately with an error message explaining the reason for the denial. If a request successfully passes all admission controllers, the API server performs a final schema validation before persisting the object to the cluster's data store. [33, 36]

## 3.2 A Deep Inspection of Role-Based Access Control (RBAC)

Role-Based Access Control (RBAC) is the primary authorization mechanism within Kubernetes, providing a declarative framework for managing permissions through objects in the `rbac.authorization.k8s.io/v1` API group. Evaluated on every API request, the RBAC model aligns with Kubernetes' core design: policies are defined as API resources that can be versioned, audited, and managed with the same workflows as other cluster components. A precise understanding of RBAC is therefore essential for enforcing the principle of least privilege, as it governs *who* (subjects) may perform *which actions* (verbs) on *which resources*, and in *what scope* (namespace or cluster-wide). The RBAC model is built on several fundamental principles. It is **additive-only**, meaning there are no explicit "deny" rules; a request is denied by the absence of an applicable rule that allows it. A key design feature is the separation of *permission definition* from *permission assignment*. This is realized through four core object kinds that work in concert: `Roles` and `ClusterRoles` define sets of permissions, while `RoleBindings` and `ClusterRoleBindings` grant those permissions to specific subjects: Users, Groups, or ServiceAccounts, based on the exact string identity asserted by the authenticator. [2]

**Role (Namespaced Permissions)** The most granular definition of permissions is the **Role**, a namespaced object that grants access to resources exclusively within its own namespace. A Role contains a list of rules, where each rule enumerates the permitted **verbs** (actions) on a set of **resources** within one or more **apiGroups**. For highly specific grants, a rule can be further scoped to individual object instances via the **resourceNames** field.

Listing 3.1. A Role granting read-only access to Pods and a single, named ConfigMap.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: team-reader
  namespace: team-a # The scope of this Role is the 'team-a'
  namespace
rules:
- apiGroups: [""] # The core Kubernetes API group
  resources: ["pods"]
  verbs: ["get","list","watch"]
- apiGroups: [""]
  resources: ["configmaps"]
  resourceNames: ["app-prod-config"]
  verbs: ["get"]
```

*Explanation.* The empty string [""] in **apiGroups** designates the core API. The second rule demonstrates the use of **resourceNames** to restrict permissions to a specific object.

**ClusterRole (Cluster-Wide or Reusable Permissions)** For authorization permissions that must operate beyond namespace limitations, the model offers the **ClusterRole**. Since it is defined as a non namespaced object, it fulfills two separate, critical functions: (i) it enables the granting of permissions specifically targeting cluster scoped resources, such as **nodes**, **persistentvolumes**, or the **namespaces** themselves; and (ii) it provides a standardized collection of permissions for namespaced resources, which can then be applied consistently across all namespaces when assigned through a corresponding **ClusterRoleBinding**.

Listing 3.2. A ClusterRole for viewing nodes, all Pods, and scaling Deployments.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: support-view
rules:
- apiGroups: [""]
  resources: ["nodes"]
  verbs: ["get","list","watch"]
- apiGroups: [""]
  resources: ["pods","configmaps"]
  verbs: ["get","list","watch"]
```

```

- apiGroups: ["apps"]
  resources: ["deployments/scale"] # A subresource
  verbs: ["get", "update", "patch"]
- nonResourceURLs: ["/healthz", "/livez", "/readyz"]
  verbs: ["get"]

```

*Explanation.* Permissions on non-API endpoints are specified via `nonResourceURLs`, while access to object facets like scaling controls is managed via `subresources`.

**RoleBinding (Namespaced Assignment)** The formal definition of permissions is kept separate from their actual assignment. The connection between a specific Role and the subjects who will receive its permissions is created by a **RoleBinding**. This namespaced object explicitly links a set of subjects to a defined Role, which consequently grants the Role's permissions to those subjects, strictly limited to the binding's own namespace. This established mechanism serves as the primary standard for delegating authority with well defined boundaries.

Listing 3.3. A RoleBinding granting a namespaced Role to multiple subjects.

```

apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-team-a
  namespace: team-a # Grants are effective only in this namespace
subjects:
- kind: ServiceAccount
  name: api
  namespace: team-a # Namespace is required for ServiceAccount
  subjects
- kind: User
  name: alice@example.com # External user identified by a string
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: team-reader # Must exist in the 'team-a' namespace

```

*Explanation.* The `namespace` field is required for `ServiceAccount` subjects but must be omitted for `User` or `Group` subjects, as their identities are cluster-global.

**ClusterRoleBinding (Cluster-Wide Assignment)** Correspondingly, the assignment of a `ClusterRole` across the entire cluster is accomplished via a **ClusterRoleBinding**. This non-namespaced object grants the permissions defined in a `ClusterRole` to subjects globally, making it the mechanism for configuring cluster administrators, global auditors, and system-wide controllers.

Listing 3.4. A ClusterRoleBinding granting cluster-wide permissions to a group.

```

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding

```

```

metadata:
  name: support-view-all
subjects:
  - kind: Group
    name: corp:support # Group string provided by an external IdP
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: support-view

```

*Explanation.* This binding makes the ClusterRole’s rules for namespaced resources effective in every namespace, in addition to enabling its rules for cluster-scoped resources.

### 3.2.1 Particular Cases in RBAC

Beyond the core objects, several nuances in the RBAC model are critical for its correct and secure implementation.

**Scoping a ClusterRole with a RoleBinding** A particularly flexible and powerful feature of the Kubernetes RBAC model is the ability for a namespaced `RoleBinding` to reference a cluster-scoped `ClusterRole`. When this pattern is employed, the permissions granted by the `ClusterRole` are not conferred globally. Instead, only the rules within the `ClusterRole` that pertain to namespaced resources are applied, and their scope is strictly confined to the namespace where the `RoleBinding` resides. Any permissions for cluster-scoped resources or non-resource URLs defined in the `ClusterRole` are disregarded in this context. This mechanism enables a highly effective administrative pattern: a central catalog of reusable roles can be defined once at the cluster level and then safely delegated for assignment within specific, localized scopes.

Listing 3.5. A `RoleBinding` scoping a `ClusterRole`’s permissions to a single namespace.

```

apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: support-view-team-b
  namespace: team-b
subjects:
  - kind: ServiceAccount
    name: dashboard
    namespace: team-b
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole # Reference a ClusterRole...
  name: support-view # ...whose permissions are now scoped to
                    'team-b'.

```

**Subresources and High-Privilege Verbs** Many sensitive operations are modeled not as top-level resources but as **subresources** or special verbs, each requiring an explicit grant.

- Interactive data-plane access to workloads is controlled via subresources such as  `pods/exec`,  `pods/portforward`,  `pods/attach`, and  `pods/log`.
- Horizontal scaling is managed through subresources like  `deployments/scale`.
- The ability for a principal to act as another is governed by the  `impersonate` verb on  `users`,  `groups`, and  `serviceaccounts`.
- The power to modify the RBAC policy itself is controlled by the  `bind` and  `escalate` verbs on  `Role` and  `ClusterRole` objects; these should be granted with extreme care.

**Aggregated Roles** To facilitate the management of complex and extensible permission sets, RBAC includes two mechanisms for role aggregation.

1. **Label-based contribution:** A  `ClusterRole` can be labeled (e.g., with  `rbac.authorization.k8s.io/aggregate-to-edit: "true"`) to have its rules automatically merged into one of the default user-facing roles like  `edit`.
2. **Explicit aggregation rule:** A custom "umbrella"  `ClusterRole` can be defined with an  `aggregationRule` that selects other  `ClusterRoles` by label. The controller then dynamically unions their rules into the umbrella role.

Listing 3.6. Two forms of role aggregation: (A) contributing to a built-in role, and (B) a custom umbrella role.

```
# (A) This ClusterRole's rules will be added to the built-in
    "edit" role.
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: app-edit-extensions
  labels:
    rbac.authorization.k8s.io/aggregate-to-edit: "true"
rules:
  - apiGroups: ["apps"]
    resources: ["deployments","statefulsets"]
    verbs: ["update","patch"]
---
# (B) This ClusterRole aggregates all rules from roles labeled
    'app.k.io/part=observability'.
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: observability-admin
```

```
aggregationRule:
  clusterRoleSelectors:
    - matchLabels:
        app.k8s.io/part: observability
  rules: [] # This is automatically populated by the controller.
```

*Explanation.* Mechanism (A) allows add-ons to extend default roles, while (B) enables the creation of centralized, composite roles from a set of modular contributor roles.

## 3.3 Network Policy: Data-Plane Segmentation

This section introduces Kubernetes `NetworkPolicy` as the data-plane complement to API-plane authorization, deciding *which Pods may talk to which other endpoints* on the data plane. At Layer 3/4 (IP and port), a `NetworkPolicy` carves the flat, fully routable Pod network into least-privilege “micro-segments”. This is the primary in-cluster control for limiting lateral movement: once a Pod is *selected* by a policy, only traffic explicitly permitted by that policy (and any others that select the same Pod) is allowed [3].

### 3.3.1 From a Flat Network to Intentional Segments

Kubernetes’ baseline is deliberately permissive: every Pod can reach every other Pod on any port, across namespaces. This maximizes out of the box operability but also enlarges the blast radius if a workload is compromised. A `NetworkPolicy` introduces intentional segmentation by flipping the default for selected Pods and directions from “allow everything” to “deny unless explicitly allowed.” Semantically, policies operate as additive allow lists: traffic is permitted only when at least one applicable allow rule matches, there are no explicit deny rules, and the order of rules is immaterial. Enforcement is not performed by the API itself but by the Container Network Interface implementation, which realizes the declarative specification in the data plane by translating it into `iptables`, `nftables`, or eBPF programs. Consequently, precise feature coverage, for example support for egress, named ports, SCTP, or port ranges, depends on the chosen plugin.

### 3.3.2 Anatomy of a NetworkPolicy

A `NetworkPolicy` is a namespaced resource in the `networking.k8s.io/v1` API group whose scope is determined by `spec.podSelector`. This selector identifies target Pods within the policy’s own namespace by labels; an empty selector (`()`) applies the policy to all Pods in that namespace. By design, policies do not select Pods across namespace boundaries; any intent to regulate cross namespace communication must be expressed within the policy’s rules rather than through the initial selection.

The application of a `NetworkPolicy` alters a Pod’s security posture in a precise manner. By default, Pods are network-permissive; however, once a Pod is selected by a policy, it transitions to a **default-deny** stance for the traffic directions specified in its `policyTypes` field (`Ingress`, `Egress`, or both). If this field is omitted, the direction is inferred from the presence of corresponding rule blocks. Consequently, all traffic in the governed directions is blocked unless an explicit rule permits it. Pods not targeted by any policy remain in the default-allow state.

Permitted traffic is defined within `ingress` and `egress` rule blocks. These rules are evaluated with **OR logic**, meaning traffic is allowed if it satisfies the criteria of *any* single rule. Within an individual rule, however, all conditions are combined with **AND logic**, requiring every constraint to be met. Each rule can specify

allowed traffic based on its peers (source or destination) and its network ports. A peer, defined in a `from` block for ingress or a `to` block for egress, can be a set of Pods (`podSelector`), Pods in other namespaces (by combining a `namespaceSelector` and a `podSelector`), or an external CIDR range (`ipBlock`). The rule can be further refined by specifying ports, including the protocol (TCP, UDP, or SCTP), a numeric port, a named port resolved from the Pod's specification, or a port range.

It is essential to understand that this enforcement occurs at the Pod level, filtering traffic destined for a Pod's IP address regardless of how the connection is established, whether directly or through a `Service`. The policy evaluates the connection against the destination port on the container itself, which typically corresponds to a `Service`'s `targetPort`. Crucially, these policies do not govern traffic originating from or terminating on the worker node's own network namespace; such traffic is managed by the underlying Container Network Interface (CNI) plugin.

Listing 3.7. Minimal, generic `NetworkPolicy` skeleton.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: <NAME>
  namespace: <NAMESPACE> # policies are namespaced
spec:
  podSelector: {} # select all Pods in this namespace (or add
    labels)
  # policyTypes can be omitted and is inferred from the presence
  # of sections below
  policyTypes: # e.g., ["Ingress","Egress"]
  - Ingress
  - Egress
  ingress: # list of ingress allow rules (OR across rules)
  - from: # list of peers (OR across items)
    - podSelector: # same-namespace Pods
      matchLabels:
        <KEY>: <VALUE>
    - namespaceSelector: # select namespaces by labels
      matchLabels:
        <KEY>: <VALUE>
      podSelector: # AND with a podSelector in those namespaces
      matchLabels:
        <KEY>: <VALUE>
  - ipBlock: # external CIDR ranges
      cidr: 10.0.0.0/8
      except: ["10.1.0.0/16"]
  ports: # optional L4 constraints
  - protocol: TCP
      port: 80 # or a named port defined in the Pod
      endPort: 90 # optional numeric range end
  egress: # list of egress allow rules (OR across rules)
  - to:
```

```
- namespaceSelector:
  matchLabels:
    <KEY>: <VALUE>
ports:
- protocol: UDP
  port: 53 # e.g., allow DNS to kube-dns CoreDNS Service
```

*Notes.* (i) Inside a single rule, peers and ports are AND-ed; across rules they are OR-ed. (ii) Cross-namespace communication is modeled via `namespaceSelector` (optionally AND-ed with a `podSelector`).

# Chapter 4

## Thesis Objective

In modern cloud environments, security posture is not a static blueprint but the emergent result of countless configurations that are vast, heterogeneous, and continuously changing. Policies governing critical layers, such as authorization, resource relationships, network connectivity, and service exposure, are authored in a decentralized manner by different teams and deployed independently via CI/CD pipelines. In Kubernetes, this fragmentation of ownership and artifacts is manifest in configuration dispersed across many object kinds (e.g., `Role`, `ClusterRole`, `RoleBinding`, `ServiceAccount`, `Deployment`, and `NetworkPolicy`), often split across namespaces. Security analysts are pushed into laborious manual routines to collate configuration, reconcile it with the running cluster, and reason across layers simultaneously. This jeopardizes foundational principles: *separation of duties* and *least privilege* can be undermined when subtle cross-file interactions unintentionally aggregate permissions, and *defense in depth* weakens when missing or inconsistent controls at one layer negate protections at others [1]. Prior work confirms that manual reviews do not scale and tend to accumulate inconsistencies over time in the absence of a common formal basis to normalize and check the cumulative effect of many small changes [9, 10, 8].

A key practical challenge originates from the reliance on *domain specific tools*, with each offering only a restricted, layer constrained view of the system's operational state. Tools engineered for network policy verification can precisely model traffic flows yet operate completely unaware of the authorization layer. They cannot determine which specific users or processes possess the authority to modify the policies or the workloads themselves. In contrast, authorization analysis utilities can comprehensively delineate all permissions (the subject action resource triad) but hold no information regarding network topology. This functional segregation generates dangerous security blind spots, because the most critical risks frequently arise from the dynamic interaction between these separate policy domains. For instance, an authorized principal with permission to modify a vital `Deployment` might introduce a new container image that, when coupled with a seemingly minor adjustment to an `Egress` policy, obtains the capability to exfiltrate sensitive data. This emergent, cross domain risk is impossible to identify using any single function tool, since its detection necessitates a holistic comprehension of how authorization permissions and network connectivity rules ultimately intersect.

The academic literature reflects and reinforces this fragmentation: connectivity research offers efficient, formally grounded reachability analyses [13, 14] but explicitly stops at the network boundary, ignoring identities and privileges. In parallel, the identity and access management (IAM) domain provides frameworks for access-control refinement and verification [37, 12] but remains topology-agnostic. Taken together, these strands motivate an integrative approach in which scalable verification demands the normalization, formalization, and automated reasoning that composes identity/authorization with reachability.

This thesis addresses that gap by proposing a unified, query-driven framework in which operators express high-level questions about the combined security posture of a Kubernetes cluster and obtain deterministic answers computed against a formal model. Queries are written in a compact, operator-oriented language grounded in subjects, actions, resources, and flows, and compiled to **SWI-Prolog** for evaluation against a small, transparent set of logical rules. To retain practical relevance while keeping scope tractable, the initial instantiation targets Kubernetes **RBAC** and **NetworkPolicy**; this focus suffices to surface cross-domain risks that elude single-purpose tools, while the underlying architecture is designed to be **modular and readily extensible** to additional policy and configuration sources. In doing so, the work contributes a single locus where topology-aware verification and authorization-aware analysis interact explicitly, offering a normalized knowledge base and deterministic cross-layer query answering that neither IAM frameworks nor reachability checkers can provide in isolation.

## 4.1 A Taxonomy of High-Level Security Queries

This section introduces the families of high-level security questions that form the foundation of this thesis. The methodological approach of this thesis does not begin with an abstract analysis of syntax or a catalog of configuration objects. Instead, our starting point was to identify the practical and urgent questions that operators and security teams face in their daily work, particularly during critical activities such as code reviews, incident response, and audits. The purpose of this section is therefore to present, on a conceptual and discursive level, the types of queries that emerged from this analysis. These queries form the conceptual foundation upon which the entire development of this thesis is based.

### High level Queries Overview

Everything begins with the simplest, most urgent question an operator can ask: **“Is this allowed?”** These fundamental checks form the atomic unit of security verification and come in two essential flavors.

The first, focused on **authorization**, directly parallels the functionality of the native Kubernetes `kubectl auth can-i` command, but with significant enhancements that will be detailed in later sections. It is used to decide if a subject holds a specific permission on a resource, answering critical questions like, *“Can the service*

account *sa-ci* update the *Deployment/frontend* in *staging*?” or, more broadly, “Someone can delete *ConfigMaps* in *webapp*?”.

The second flavor applies the same decisive logic to **communication**, verifying network reachability under the current set of policies. This allows operators to determine answers to connectivity questions such as, “Can pods with label *app=frontend* reach pods with *app=backend* on *TCP/5000*?”.

Answering “what” is possible naturally leads to the question of “how.” To understand the source of a permission, we need to peel back a layer of abstraction and examine the declared relationships. This is the job of **direct role-binding queries**. These queries don’t expand or infer effective permissions; they inspect the explicit link between a subject and a role, allowing an operator to audit the grant itself from either direction. One can ask from the subject’s perspective, “Is user *alice@corp* directly bound to *ClusterRole/view*?”, or from the role’s, “List all subjects directly bound to *role/pod-reader* in *webapp*.”.

With these foundational relationships clarified, the next challenge is to see the complete picture. In a real environment, configuration is scattered across **YAML** files. Manually cross-referencing these sources is slow and error-prone. **Listing queries** solve this by providing a flexible way to enumerate desired values, such as actions, permissions, subjects, or network connections based on a specific condition or starting entity. Instead of being limited to a single perspective, these queries allow the direction of the inquiry to be inverted based on what the operator needs to discover. For instance, an operator can start with a subject and ask the system to list all of its capabilities, as in, “Show all actions that the *ServiceAccount sa-ci* can perform.” Conversely, they can start with a known permission and ask for an enumeration of all subjects who hold it, answering the critical question, “Which subjects have the permission to *create Pods* in the *prod* namespace?” This flexibility extends to inspecting the permissions granted by a specific role or, in the network domain, querying for a complete list of a workload’s allowed traffic, such as, “Enumerate all allowed ingress connections for pods with the label *app=frontend* in the *webapp* namespace.”

Once we have these complete inventories, we can start to compare them. This is often where subtle but dangerous configuration issues are found. **Permission-delta queries** are designed for this comparative analysis. They expose privilege creep, unintended parity between different teams, or policy drift over time by answering questions like, “What permissions does *user1* have that *user2* does not?” or “Where do the permissions for the *devs* and *ops* groups overlap in the *payments* namespace?”.

Knowing who has what is only half of the work. The most significant risks in complex systems often emerge not from a single bad permission, but from the toxic interaction of several seemingly benign ones. This is where the focus shifts from visibility to active enforcement.

The most critical queries in this family are for finding **dangerous combinations**. They assert that certain properties must not hold together for the same principal or within the same scope. This is the primary tool for technically enforcing principles like Separation of Duties, by flagging scenarios where authorization is paired with risky connectivity, or where multiple sensitive verbs are held by a

single actor. Examples include enforcing invariants like, “*No subject may update any `Deployment` in `payments` and have egress to `0.0.0.0/0` on `TCP/443`,” or detecting risky capabilities such as, “*Does `user1` `Create` a `Pod` in `prod` and also `Create` into  `pods/exec`.”**

Beyond quantitative limits, a robust security posture also requires qualitative controls that define explicit trust boundaries. This is where **cardinality queries** translate governance goals into verifiable numbers. They put a “budget” on risk by ensuring that powerful capabilities are not too widely distributed, answering questions like, “*Are there no more than three subjects who can update a `Deployment` in `prod`?*” or “*Is it true that at most one role per namespace grants the permission to create `Pods`?*”

Alongside these quantitative bounds, some rules are absolute and must be strictly enforced. This is the role of **restricted permission** and **restricted action queries**, which provide the tools to define and verify these non-negotiable lines. The first type, a **restricted permission query**, functions as a strict **allowlist**. The mechanism is straightforward: an operator specifies a sensitive property—such as a permission or a network path—and provides an explicit list of approved subjects or pods. The query then verifies that no principal or pod *outside* this designated list possesses the specified capability. For RBAC, this allows enforcing rules like, “*Can someone outside the `cluster-admins` group delete namespaces?*” In networking, the same model can lock down critical communication paths, for example, by verifying that “*Can only pods with the label `app=api-gateway` can initiate connections to pods with the label `app=user-db` on `TCP` port `5432`.”* Complementing the allowlist model, the **restricted action query** creates what can be thought of as a functional **guardrail** for a specific subject or policy. Instead of defining *who* can access a resource, this query defines *what* a specific subject is allowed to do, ensuring its capabilities are a strict subset of an approved action list. This is crucial for locking down automated tooling and service accounts, for example, with a policy like, “*Ensure the `ci-bot` service account can only perform `get`, `list`, and `watch` actions on `ConfigMap` objects in the `builds` namespace.*” The same logic applies to network policies, where it can be used to verify that a policy is strictly unidirectional, as in, “*Verify that the `Pod` named `np-frontend` is strictly `ingress-only`.”*

Ultimately, a strong security posture requires managing the **entire configuration lifecycle**. As environments change, obsolete policies and roles are often abandoned, resulting in unnecessary complexity and potential vulnerabilities.

**Zombie-role detection** addresses this through vital cleanup by identifying `Role` and `ClusterRole` objects that lack any active subject bindings. By highlighting these unused artifacts, this process allows teams to efficiently answer questions such as, “*Have any unbound `ClusterRoles` remained following recent migrations?*”. Eliminating this *zombie* configuration minimizes clutter, prevents the accidental assumption of stale or overly permissive roles, and ensures the privilege model stays clear and auditable over time.

The taxonomy of queries presented in this section represents the foundational requirements that have driven the technical work of this thesis. The following table

provides a condensed summary of these query families, categorizing them by domain and presenting their core intent alongside a representative example for each.

Table 4.1: Operator-friendly query families with concise intent and examples.

Category	Query Name	Intent	Example
RBAC	<b>Can I? (Authorization)</b>	True/False check on a subject-verb-resource triple.	can sa-ci@builds update deployments@builds #frontend?
RBAC	<b>Who is bound to what?</b>	Inspects Role/ClusterRole bindings.	Is user alice@corp bound to clusterrole/view?
RBAC	<b>Explore &amp; List</b>	Enumerates subjects, roles, actions, or resources.	List subjects that can create Pods.
RBAC	<b>Permission Delta</b>	Differs or asserts equivalence between principals or roles.	What can user1 do that user2 cannot?
RBAC	<b>Cardinality Constraint</b>	Puts a cap on subjects with a powerful permission.	$\leq 3$ subjects may update deployments.
RBAC	<b>Allowlist Only</b>	Ensures only approved actors hold a sensitive privilege.	Only cluster-admin can delete namespaces.
Network	<b>Can it talk? (Reachability)</b>	Checks L3/L4 connectivity under current policies.	can Pod@app#frontend communicate to Pod@app#backend on TCP/5000?
Network	<b>Namespace Isolation</b>	Verifies or enforces namespace network boundaries.	No ingress and egress connection for pod with label app=demo.
Network	<b>External Exposure Check</b>	Controls or audits egress to the public internet.	No Pods have egress to 0.0.0.0/0 on TCP/8080.
Network	<b>Directionality</b>	Guarantees a workload is ingress-only or egress-only.	Verify that Pod named np-frontend is ingress-only.
Network	<b>Connectivity Cardinality</b>	Caps the number of allowed peers for a workload.	$\leq 1$ egress peer per backend pod on TCP/5432.

Category	Query Name	Intent	Example
Network	<b>Explore &amp; List</b>	Enumerates endpoints or allowed network flows.	List all ingress connections from sources with label <code>app=frontend</code> .
Cross-Domain	<b>Dangerous Combinations</b>	Finds risky intersections of RBAC + network paths.	No <code>serviceaccount</code> may update <code>deployments</code> in <code>prod</code> and Pods in the same namespace do not have egress to internet.
RBAC (Role only)	<b>Zombie Roles</b>	Finds <code>Roles/ClusterRoles</code> not bound to any active subject.	List roles left over from past migrations.

The taxonomy of queries presented in this section is therefore more than a conceptual catalog; it represents the foundational requirements that have driven the technical work of this thesis. Each query family, derived from real-world operational needs, served as a primary driver for the design of the system’s architecture. The central challenge was to create a system capable of answering sophisticated questions that range from deep inquiries within a **single policy domain** to complex checks that **span across them**, all while adhering to two key principles: the query language exposed to the operator must be direct and expressive, and the underlying architecture must be modular and readily extensible to new policy domains.

With these requirements established, the following sections will detail the architecture designed to meet them. We will present each component of the system in turn, explaining its specific function and how it contributes to the overall goal of providing a unified, formally grounded framework for cloud security policy verification.

# Chapter 5

## Architecture Description

Building upon the taxonomy of operator-centric security questions established in the previous chapter, this chapter details the architecture designed to answer them. The fundamental challenge is to translate those high-level, often cross-domain, security questions into a system that can provide formal, deterministic, and verifiable answers. To achieve this, the architecture is founded on a set of core design principles aimed at creating a single, extensible, and rigorous framework for security posture verification.

The first and most critical principle is the adoption of a **unified data model**. In production, security posture emerges from a collection of heterogeneous policy types. Our architecture confronts this complexity by normalizing disparate configuration sources, such as Kubernetes `RBAC` and `NetworkPolicy` objects, into a common, consistent logical model. This creates a single locus of truth where questions can be evaluated compositionally, enabling the analysis of both single-domain and cross-domain properties without needing to reconcile incompatible formats by hand.

This unified model is interrogated through a **high-level query language**. This language serves as a stable abstraction layer, shielding operators from the implementation details and syntactic of individual policy formats. The vocabulary is intentionally lean and grounded in familiar operational concepts like subjects, actions, resources, and network flows. This allows operators to formulate direct and expressive questions about what is truly possible within the cluster, receiving clear answers such as yes/no decisions, complete enumerations of satisfying cases, or quantitative count evaluations.

To guarantee long-term upkeep and overall viability, the system is engineered for **modularity and high extensibility**. The architecture establishes a clear division between a stable central core (which governs the query language semantics and the core evaluation logic) and a suite of pluggable **adapter components**. The function of each adapter is to extract concrete facts from a specific, domain-related source and translate them into the unified data model used by the system. This design choice makes extensibility a fundamental feature. While the initial focus is on verifying `RBAC` and `NetworkPolicy`, the structure readily allows for the future addition of new adapters (for example, those managing admission controls, Pod Security Standards, or external Identity and Access Management (IAM) systems)

without ever having to alter the central evaluation logic or how operators formulate their verification questions.

Finally, these principles combine to **reduce cognitive load and shorten verification cycles**. By providing a single, coherent interface for posing complex security questions, the system speeds up change reviews and sharpens the boundaries of responsibility between teams. It enables the automation of recurring controls, such as checking for toxic permission combinations or enforcing permission allowlists, and facilitates the hygienic maintenance of security posture by making it easier to identify issues like orphaned roles or privilege sprawl.

With these architectural goals in mind, the following sections will present a detailed description of each component of the system. We will examine their specific functions and explain how they work together to provide a formally grounded framework that turns operator-centric questions into verifiable security insights.

## 5.1 Architecture Description

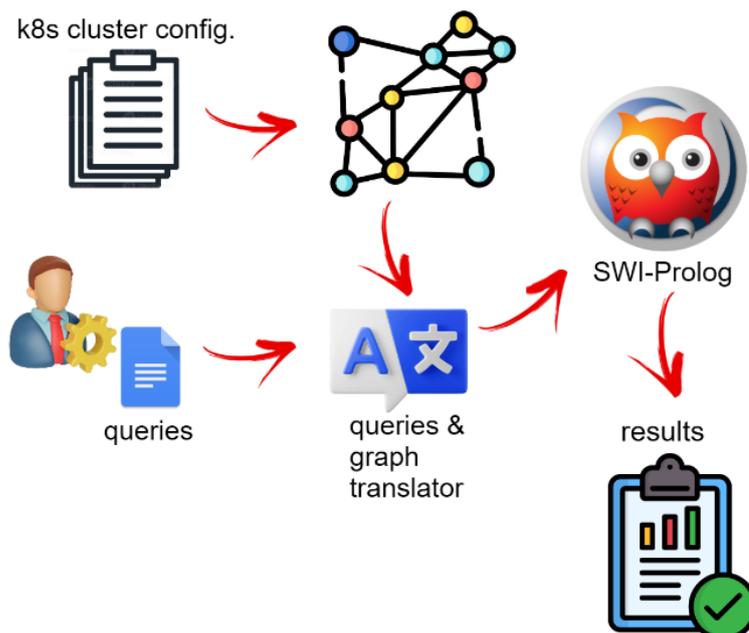


Figure 5.1. High-level data flow of the security query architecture.

The system’s architecture, visualized in Figure 5.1, is designed as a straightforward pipeline that translates an operator’s high-level security question into a formal, verifiable answer. This workflow begins with two primary inputs: a snapshot of the **Kubernetes cluster configuration** and the **queries** formulated by an operator. These inputs converge at the **queries & graph translator**, which normalizes the configuration into a unified model and translates the query into a logical goal. This pair is then passed to the reasoning core, powered by **SWI-Prolog**

(whose declarative nature is exceptionally well-suited for graph traversal and logical inference) which performs the formal verification. Finally, the outcome is formatted into actionable **results**, such as boolean decisions or enumerations.

This architectural design inherently supports **cross-domain questions** and is **extensible** by design. Having outlined the end-to-end process, we will now examine each stage of the pipeline in detail, beginning with the foundational input: the cluster configuration snapshot.

## 5.2 Architectural Components in Detail

This section provides a detailed examination of the primary components of the architecture. Components are designed with well-defined responsibilities to promote modularity, facilitate verification, and support extensibility.

### 5.2.1 Cluster Representation: Scope, Content, and Rationale

The first architectural element is a faithful representation of the cluster configuration, conceived as a coherent snapshot that isolates the semantic state at a given instant. For the purposes of this thesis, the snapshot comprises three coordinated components: a complete enumeration of the resources present in the cluster, the authorization policies defined through RBAC, and the network connectivity policies defined through NetworkPolicy. This triad provides the vocabulary of objects and actions, the assignment of capabilities to identities, and the intentional topology of permitted flows. .

The resource component includes every resource in the cluster, subresources included. Conceptually, each resource is represented as a single logical record. The record is identified by the canonical path that Kubernetes itself uses to denote the resource and, when applicable, its subresource. A path can be expressed abstractly as `/apis/<group>/<version>/namespaces/<ns>/<kind>/<name>/<subresource>` or `/api/v1/namespaces/<ns>/<kind>/<name>/<subresource>` in the core group. For concreteness, one may think of `/apis/apps/v1/namespaces/webapp...  
../deployments/frontend/scale` as a typical example that names a deployment's *scale* subresource. Alongside the path, the record includes the set of verbs that can be associated with an RBAC permission over that resource or subresource, and the set of labels if they are present.

Verbs and labels are not incidental embellishments. Verbs delimit the action domain of each resource and subresource, ranging from general capabilities such as reading and listing to sensitive capabilities that expose control surfaces, for instance process execution in a Pod or dynamic scaling of a controller. Retaining the verb domain is essential whenever one wishes to pose expressive, theory-level questions that depend on what actions are even possible on a given object before asking who is allowed to perform them. Labels, in turn, provide the language for speaking

about sets of objects rather than about isolated instances. NetworkPolicy relies on label selectors to define groups of Pods and Namespaces, therefore a correct understanding of policy scope depends on knowing how labels are distributed across the resource universe. Moreover, labels support a monotonic style of governance, since newly created objects that share labels and domain inherit the intended policies without revising the rules. Taken together, the path, the verb domain, and the label set furnish all information needed to support a wide range of queries that span the two policy domains studied in this thesis, namely authorization and connectivity.

The RBAC component consists of the cluster’s authorization configuration in its canonical declarative form. It includes roles and cluster roles, which aggregate permissions as pairs of verbs and resources possibly constrained by names or API groups, as well as role bindings and cluster role bindings, which assign those aggregated capabilities to subjects such as users, groups, and service accounts within namespaced or cluster-wide scope. This corpus contains the entirety of the information required to reason about who may perform which actions and where in the cluster that power holds.

The NetworkPolicy component consists of the cluster’s connectivity configuration in its canonical declarative form. It includes all policies that describe ingress and egress relations between classes of endpoints, which are defined by `podSelector` and `namespaceSelector`, together with transport qualifiers such as ports and protocols and, when relevant, external address ranges expressed as CIDR blocks. This corpus contains the entirety of the information required to reason about which communications are admissible among the classes of objects defined in the resource component.

This first layer of the architecture maintains flexibility and ease of extension. The three components share a common semantic grammar based on labeled sets, capability domains, and declarative relations. As a consequence, accommodating a new policy domain follows a straightforward strategy. One introduces a new input corpus that captures the domain’s policies in a canonical declarative shape aligned with the existing grammar, and the snapshot expands without disturbing prior reasoning. In this way the representation remains conceptually uniform, analytically powerful, and amenable to rigorous verification across evolving requirements.

## 5.2.2 Graph Representation of Cluster State and Policy

This section presents the second component of the architecture. Its objective is to assemble the information produced by the cluster snapshot into a single, coherent mathematical object that supports uniform reasoning across policy domains. The snapshot already provides a complete catalogue of resources with canonical paths, verb domains, and labels, an authorization corpus that specifies subjects, roles, and bindings, and a connectivity corpus that specifies ingress and egress relations between labeled classes of endpoints. The graph component turns these corpora into a unified structure where objects and relations coexist as first-class elements. The unification serves three aims: it offers a single view of policy in which authorization, connectivity, and configuration attach to the same entities; it admits native extensibility by introducing new node or edge kinds without disturbing prior

structure; it preserves local semantics because node and edge types and attributes encode a significant part of the intended meaning.

## Graph Primitives and Typing

We model the cluster and its policies as a typed, directed graph  $G = (V, E)$  with explicit kinds for vertices and edges. A total function  $\tau_V : V \rightarrow \mathcal{K}_V$  assigns each vertex a kind, and  $\tau_E : E \rightarrow \mathcal{K}_E$  assigns each directed edge a kind. The snapshot assumption guarantees that  $V$  and  $E$  are finite and refer to the same logical state. In this thesis we distinguish, at a high level, two broad families of vertex kinds. First, *resource vertices* represent Kubernetes resources and their subresources, both namespaced and cluster-scoped; they provide the structural carrier of the configuration and are the anchors to which policies ultimately refer. Second, *policy-domain vertices* capture entities introduced by the policy layers themselves, such as principals in the authorization domain (e.g., users and groups) or auxiliary endpoints in the connectivity domain (e.g., external address spaces like CIDR blocks). This separation allows us to reason uniformly about what exists (resources) and about who or what participates in policy relations (domain-specific entities), while deferring fine-grained taxonomy to the subsequent subsections.

Edges encode typed relations among these vertices. We use `CONTAINS` to express structural inclusion along the resource hierarchy, `GRANTS` to express policy entitlements from policy-bearing vertices to their resource targets, `BINDS` to associate principals with policy-bearing vertices in the authorization domain, and `CONNECTS` to record admissible communication flows in the connectivity domain. When a relation is qualified, edges carry finite attributes native to the domain, such as sets of actions for entitlements or transport qualifiers for connectivity.

## Backbone: A Hierarchical Structure for Kubernetes Resources

At the core lies a tree-shaped subgraph whose edges are of kind `CONTAINS`. The root is a unique vertex labeled `cluster`. This backbone encodes the logical containment hierarchy of the cluster in a way that separates kinds from instances and instances from subresources, thereby preserving both abstraction and precision. Figure 5.2 provides a simplified visualization of this structure, starting from the `Root` and branching out to `Buckets` and concrete instances like `ns1` and `Frontend-Pod`.

The hierarchy proceeds in layers. Immediately under the root, a bucket vertex is introduced for each cluster wide resource kind and a dedicated bucket for namespaces. Buckets represent kinds and not concrete objects. Each bucket contains its instances. Hence, `Bucket[Node]` contains concrete node vertices such as: `Node/my-node`, and `Bucket[Namespace]` contains concrete namespace vertices: `namespace/my-namespace`. Namespace instances are the only vertices that further expand the hierarchy. Under a namespace vertex, there is one bucket per namespaced resource kind, and each such bucket contains the instances of that kind within the namespace. When a resource exposes subresources, these appear as child vertices under the corresponding instance. For example, in Figure 5.2, `Pod/Exec` is shown as a subresource of `Frontend-Pod`. Buckets only contain instances of their

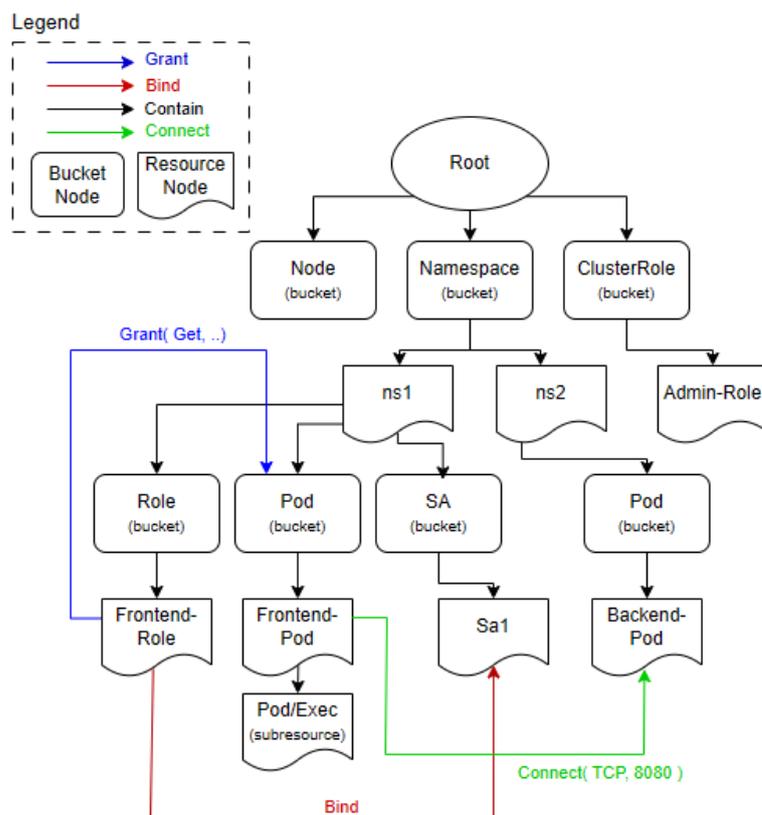


Figure 5.2. Example of the unified, typed graph model showing structural containment, RBAC overlays (GRANT, BIND), and NetworkPolicy connectivity (CONNECT).

kind, and namespace instances only contain buckets for namespaced kinds. This disciplined structure yields canonical places where policies can later attach.

## RBAC as an Authorization Overlay

The Role-Based Access Control (RBAC) model is implemented as an overlay on the graph’s existing structural backbone, leveraging the *Role* and *ClusterRole* vertices (see blue and red edges in Figure 5.2). In this model, policy subjects (or principals) are represented by identity vertices. Service accounts correspond to standard resource instances already present in the graph, whereas users and groups are treated as synthetic vertices because they do not exist as first-class Kubernetes objects. This RBAC overlay introduces two fundamental typed relations: GRANTS (blue edges), which links a role to the permissions it confers, and BINDS (red edges), which associates a subject with a role.

The semantics of a GRANTS relation are determined by two independent factors: the granularity of its target and the scope of the role. Regarding granularity, a grant can target either a specific resource instance or an entire class of resources, represented by a “bucket” or kind carrier vertex. Permissions are defined by a set of verbs. When a grant targets a resource bucket, its permissions are inherited by all instances within that bucket through a subsumption mechanism that follows

the CONTAINS relationship. Optional name filters can be applied to restrict such a bucket-level grant to a specific subset of resources without changing its fundamental type or direction.

The scope of a grant is fundamentally determined by its role type. A namespaced *Role* is only valid and effective within the specific graph region that corresponds to its declared namespace. Conversely, a *ClusterRole* maintains a global operational range. For resources that are cluster scoped, its effect applies throughout the entire cluster. For namespaced resources, a *ClusterRole* furnishes a consistent set of permissions that are enforced uniformly across every single namespace. This global application is realized compositionally, by leveraging the inherent hierarchical structure of the graph, instead of relying on the generation of auxiliary or “derived” roles for each namespace.

The BINDS relation serves as the bridge connecting principals to roles. A namespace-scoped binding composes with the permissions from its referenced role, restricting their effect to that specific namespace. A cluster-scoped binding performs a similar composition but without any namespace constraint. The common pattern where a namespaced binding references a cluster-scoped role is handled by this same compositional principle: the broad entitlements of the *ClusterRole* become effective only within the limited scope of the binding. Thus, the final permission set is not derived from additional structures but emerges from the scope-sensitive composition of the BINDS, GRANTS, and CONTAINS relationships.

Authorization entailment follows directly from these ingredients. Writing  $\text{authorized}(s, v, o)$  for the derived relation “subject  $s$  may perform verb  $v$  on object  $o$ ,” we have

$$\text{authorized}(s, v, o) \text{ iff } \exists r, t. (s \xrightarrow{\text{BINDS}} r) \wedge (r \xrightarrow{\text{GRANTS}(v)} t) \wedge (t \preceq o),$$

where  $\preceq$  is the subsumption induced by CONTAINS and scope. Subsumption enforces the usual locality rules: a grant to a bucket applies to all instances it contains; a grant to a specific instance applies only there; a namespaced grant applies exclusively within its namespace. In this manner, ontology (what exists) and authority (what is permitted, and for whom) remain cleanly separated yet composable, and the overlay retains the same discursive, uniform semantics that governs the rest of the graph.

## NetworkPolicy as a Connectivity Overlay

The connectivity layer enhances the structural backbone of the graph by introducing a typed relation that models admissible network communications. Specifically, a **NetworkPolicy** is translated into directed edges of the kind CONNECTS (green edges in Figure 5.2). Each edge of this type represents a permitted data flow, oriented from a designated *source* set of endpoints to a *destination* set, and is further qualified by transport-level attributes such as protocol and destination port. By normalizing both ingress and egress rules into this uniform source-to-destination orientation, the connectivity overlay ensures a consistent traversal semantic for all network reachability queries.

Endpoint selection is inherently driven by classes and based on labels. A policy defines the sets of communication endpoints using selectors that operate on both namespaces and Pods. Specifically, a namespace selector delineates a particular area within the graph’s backbone, whereas a Pod selector identifies a subgroup of Pod vertices within that area that fulfill a specified label predicate. When a policy rule references **entire namespaces**, the CONNECTS relation is anchored at the corresponding namespace vertex level. Conversely, when a rule targets **specific pods**, the relation is attached at a finer level of granularity: either to the carrier vertex representing the Pod kind within that namespace or directly to individual Pod instance vertices, should the policy’s semantics require such precision. All clauses contained within a selector compose conjunctively, and their interpretation is **extensional**, meaning set membership is determined exclusively by evaluating the labels present in the current state snapshot.

Communication with entities outside the cluster is incorporated into this formalism by representing external address spaces (for example, CIDR ranges) as dedicated vertices within the graph structure. Consequently, the CONNECTS relation is freely applicable between internal vertices and these external endpoint representations. This modeling decision effectively internalizes extra cluster reachability, transforming it into a primary structural element that can be queried, composed, and constrained using the identical analytical machinery applied to intra cluster traffic flows.

The combination of policies is **additive** concerning permitted flows. The network overlay records only *positive* allowances: a flow exists if it is admitted by some policy and is absent otherwise. Therefore, the connectivity semantics realized in the graph matches the union of all admissible relations induced by the individual policies present in the snapshot. Any default behavior that operates outside the policy model (for example, defaults that apply to endpoints not managed by any explicit policy) is treated as an external assumption and is not represented by CONNECTS edges unless a policy explicitly admits it.

We perform reasoning on top these base edges by introducing straightforward derived predicates. Let  $\text{connects}(x, y, \pi)$  denote the condition “there is an allowed flow from  $x$  to  $y$  with transport qualifier  $\pi$ ” (which includes, for instance, the protocol and destination port). From this, we define  $\text{reaches}(x, y)$  to hold whenever a path exists from  $x$  to  $y$  that follows one or more CONNECTS edges. Two specific variants are particularly useful:

- *Protocol agnostic reachability*: Disregard  $\pi$  and determine solely whether any path exists.
- *Protocol aware reachability*: Require that every hop along the path satisfies a constraint on  $\pi$  (for example, that every single hop is `tcp:443` or belongs to an authorized set of protocols and ports).

This setup has two desirable properties. First, it is *monotone*: adding a new object that shares the same labels can only add new witnesses for a true connectivity claim, never invalidate it. Second, it is *compositional*: answers depend only on the snapshot and on the union of policy-induced edges, so connectivity reasoning aligns cleanly with the authorization layer and the rest of the graph model.

## Semantic Coherence and Extensibility

The three kind of edges serve different purposes but work well together. `CONTAINS` gives the structural hierarchy. `GRANTS` expresses permissions. `CONNECTS` expresses allowed traffic. By composing relations with `CONTAINS` we obtain two effects: *subsumption* (a statement made for a bucket also holds for the objects it contains) and *scoping* (a statement is restricted to the region picked out by a vertex such as a namespace). Authorization arises by composing `BINDS` with `GRANTS`: a subject is authorized exactly when it is bound to a role that grants the required verb to a target that contains the object of interest. Connectivity over classes arises by composing `CONNECTS` with `CONTAINS`: a class-to-class allowance induces allowances for the members of the source class and toward aggregates that contain the destination.

Because both node and edge types are explicitly defined, the model is straightforward to extend. To incorporate a new policy domain, one simply introduces the corresponding vertex kind along with a new edge type that has a clear local definition. For instance, an admission control domain would require adding a specific policy vertex and a `VALIDATES` edge, leaving the established authorization and connectivity overlays entirely unmodified.

**Summary** The typed, directed graph furnishes a single semantic substrate for cluster state and policy. The structural backbone captures ontology, the authorization overlay captures authority, and the connectivity overlay captures admissible flows. Logical queries then operate uniformly over this substrate, enabling boolean checks, witness extraction, counting, and multi hop reasoning, while remaining faithful to the abstractions that Kubernetes exposes.

### 5.2.3 High-Level Operator Queries

Having fixed the cluster snapshot and its graph-based unification as our common substrate, the second input to the system consists of the operator's high-level queries. These queries complement the snapshot by specifying what property of configuration and policy should be verified at a given time, thereby linking operator queries to the unified model introduced earlier. They are intentionally decoupled from any single policy format and are always interpreted against the graph, so that one and the same query template can range over authorization, connectivity, and any future policy domain introduced into the model. In short, the snapshot says *what exists*, the graph organizes it, and the queries state *what must hold* over that organization.

The query layer serves as a bridge by remaining *declarative*, *extensible*, and *granularity aware*. A declarative design asks for properties to be verified rather than procedures to compute them, which matches the logical reading of the underlying graph. Extensibility allows the vocabulary to expand to new domains without changing the core evaluation engine. Sensitivity to granularity lets an operator shift smoothly between concrete artifacts and classes. When details are available, a query can name specific objects through canonical paths, for instance a particular Pod

instance or subresource. When only the class matters, the same language can rely on label predicates and scope qualifiers, such as “all Pods in a given Namespace” or “all resources with a given label pair.” In this way the language supports both precise audits and higher level checks that avoid enumerating instances.

Because heterogeneous policy relationships reside within a single graph structure, a single query can traverse several distinct policy domains without requiring any auxiliary integration machinery. A typical composition combines authorization rules with connectivity rules; for example, one could ask whether a subject possessing permission to create a resource could consequently enable a network flow toward an external address space under specific transport constraints. Another useful composition connects role assignments directly to network reachability, facilitating reasoning about potential lateral movement within the cluster.

The available query vocabulary expands simply by adding new policy domains. A new adapter contributes its unique vertex kinds and edge kinds, along with a small, carefully typed set of corresponding predicates and selectors, all while leaving the core evaluation semantics untouched. Queries can then operate across these new relations in precisely the same manner they already operate across RBAC and NetworkPolicy data. This preserves operator familiarity while admitting additional sources of truth. Consequently, the query layer remains a stable, declarative interface that scales harmoniously with the underlying model and continues to formally connect operator questions to verifiable properties of the cluster.

#### 5.2.4 Translator: From Unified Graph to Logic Program

This component is responsible for materializing the cluster’s unified graph structure into a logical program, against which it then evaluates administrator queries. **SWI-Prolog** serves as both the target reasoning formalism and the execution engine. The internal translator maps all vertices, edges, and their associated finite attributes into a concise logical theory, subsequently rewriting the operator’s high-level question into a correctly typed logical goal. Query evaluation is then delegated to Prolog’s built-in proof procedure, which determines the goal’s status against the fixed policy snapshot and, where possible, enumerates the corresponding witnesses. This entire translation process is semantics-preserving: it introduces no new policy, omits none of the existing policy, but instead faithfully recasts the established structure and queries for the purpose of deterministic, formal verification.

### A Brief Overview of SWI Prolog

Prolog is a logic programming language grounded in a fragment of first order logic known as Horn clauses. Its hallmark is the declarative reading. A program denotes a set of logical statements about a world, and computation consists in asking whether a query follows from those statements. In this sense, execution is proof search.

At the core of the language are three foundational notions that together form a knowledge base. The first notion concerns **terms**, which are the data structures

of the language. A term can be an atom that names a symbol, a number that represents a quantity, a variable that serves as a placeholder to be instantiated, or a compound term that applies a functor to subterms in order to build a structured object. On top of this vocabulary, one asserts **facts**. A fact is an unconditional ground statement that records a basic relationship between terms. One also states **rules**. A rule is an implication whose head, namely the conclusion, holds whenever the body, namely a conjunction of subgoals, holds. Facts therefore provide the extensional base of knowledge, whereas rules supply intensional definitions that derive new truths from existing ones.

When a query is posed, the inference mechanism attempts to prove it through goal directed **resolution**. Operationally, the engine selects a subgoal and tries to match it against the head of a fact or a rule. If the engine chooses a rule, it replaces the subgoal with the body of that rule. The procedure continues until all subgoals are discharged. Matching is driven by **unification**. Unification is an algorithm that computes the most general substitution that makes two terms identical, and it binds variables consistently across the entire derivation. When a chosen path fails, the engine performs **backtracking**. It returns to the most recent choice point and explores an alternative. This depth first search, which combines resolution with unification and systematic backtracking, yields not only a yes or no answer but also, when relevant, concrete **witnesses** in the form of variable bindings that satisfy the query. In finite and snapshot like settings, the procedure can enumerate all justifications for a claim, or it can refute the claim by the absence of any successful derivation.

Two additional aspects deserve emphasis. First, for pure Horn clause programs, SLD resolution is sound and complete with respect to the least Herbrand model. Every computed answer corresponds to a logical consequence, and every logical consequence can in principle be found by the procedure. Second, practical Prolog systems, including SWI Prolog, strengthen the core language with efficient indexing, constraint solvers, a module system, and extensive libraries. These additions support concise encodings without sacrificing performance on real world datasets.

## Why Prolog for This Thesis

Prolog is a natural fit for the present architecture on both conceptual and methodological grounds. The unified cluster model is relational by construction, since structural containment, permission grants, role bindings, and network reachability are all relations over a finite set of entities, and the Horn clause core of Prolog maps these relations directly to a small and readable logical theory whose predicates and arguments mirror the labels of vertices and edges in the graph. Central notions admit succinct inductive definitions, including containment along ownership hierarchies, authorization derived by composing bindings with grants, and multi hop connectivity, and in Prolog these definitions appear as rules whose structure follows the mathematical clauses specified at the graph layer, which supports clean proofs, straightforward extensions, and transparent inspection of intermediate derivations. Each administrator query is evaluated by constructing a mathematical proof relative to the logical theory extracted from a fixed snapshot, where a positive outcome corresponds to a successful derivation and a negative outcome corresponds to the

absence of any derivation in that finite model; the result is deterministic and auditable, aligning with the goals of rigorous verification, and the variable bindings returned by the engine serve as explicit evidence that functions as witnesses for the reported conclusions. The introduction of an additional policy domain, for example service exposure or data flow constraints, amounts to adding well typed predicates and a small number of rules, so existing definitions remain intact, existing queries continue to operate, and the operator's style does not change, which matches the architectural objective of an extensible core with pluggable layers. Finally, SWI Prolog provides mature support for modules, tabling for the termination of suitable recursive definitions, attributed variables and constraints, efficient term indexing, and robust foreign function interfaces, thereby encouraging compact encodings of finite snapshots, delivering predictable performance on real workloads, and integrating smoothly with the surrounding tooling.

### 5.2.5 Result and Operator Presentation

The final stage of the pipeline translates formal proofs into clear, actionable results for an operator. A complete snapshot of the cluster state and its associated policies is first normalized into a unified, typed graph. High-level queries are then compiled into logical goals against this graph representation. Subsequently, a translator renders both the graph and the logical goals into a Prolog theory, which is then evaluated by a reasoning engine. A core principle of this process is that each transformation is semantics-preserving; objects and relations are not approximated or reinterpreted, but are merely recast into a format amenable to formal analysis. The final step synthesizes the outcome of the proof and presents it in a concise format that does not require the operator to understand the underlying policy syntax.

The structure of the answer is tailored to the intent of the original query, ensuring that results are compact and comparable across different runs. For queries that function as a gate, the system returns a *boolean verdict*. When the query requires evidence, the output consists of *witnesses*: the concrete subjects, resources, or communication flows that satisfy the queried property. For queries that involve a *threshold on cardinality*, the result is also *boolean*. In this case, the system first identifies and counts the witnesses and then evaluates the comparison specified in the query ( $<$ ,  $>$ , or  $=$ ). It returns *true* if the inequality holds and *false* otherwise.

With this chapter, we have presented the proposed architecture, from the unified data model to the end-to-end flow that carries cluster configurations to formally verified answers. The next two chapters complete the picture: the first introduces the query language, specifying its *syntax* and *accepted semantics*; the second presents the *proof-of-concept* (PoC), including implementation details and the results obtained.

# Chapter 6

## Language Description

This chapter specifies the query language operators use to verify cluster policies, translating the abstract question patterns from Section 4.1 into executable statements. We first define the language’s syntax, then formalize its semantics and conclude with a series of worked examples.

### Chapter Overview

This chapter formally specifies the operator-facing query language utilized for verifying Kubernetes policies across both **authorization** and **connectivity** concerns. Section 6.1 commences by introducing the core RBAC value domains (covering resources, actions, permissions, subjects, and roles) along with the system’s uniform identifier syntax. Section 6.2 then defines the NetworkPolicy value domains (protocols, ports, and network endpoints), reusing the general resource notation while specializing it for selections based on Pods and IP addresses. Subsequently, Section 6.3 formalizes the structure of predicate queries, incorporating logical quantifiers and negation, and presents a semantic validation matrix that restricts the set of meaningful root predicate combinations. Section 6.4 elevates this predicate structure to function queries, which return sets of results, featuring strongly typed selectors and explicit aggregation rules. Cardinality queries are presented as an extension, augmenting function queries with numeric constraints. The chapter concludes by detailing useful discovery patterns and providing a comprehensive table of end-to-end examples that translate common natural language security questions into their corresponding queries within the defined language.

### 6.1 RBAC Value Domains

This section *defines* the RBAC-side value domains used by the language. We adopt a bottom-up approach to build the necessary concepts. We first define the foundational objects of *resources* and the *actions* that can be performed on them. We then use these to introduce the construct *permissions*, which links actions to resources. Finally, we define the principals to whom permissions are granted, *subjects*, and conclude with *roles*, which are named collections of permissions.

### 6.1.1 Resource

The fundamental objects upon which policies are evaluated are *resources*. To define the set of all resources,  $R$ , with greater precision, we partition it based on scope. First, we define the set of *namespaced resources*,  $R_{\text{ns}}$ , as a subset of the Cartesian product of namespaced kinds ( $T_{\text{ns}}$ ), namespaces (**NS**), and names (**Name**):

$$R_{\text{ns}} \subseteq T_{\text{ns}} \times \text{NS} \times \text{Name}$$

Second, we define the set of *cluster-scoped resources*,  $R_{\text{cl}}$ , which lack a namespace component:

$$R_{\text{cl}} \subseteq T_{\text{cl}} \times \text{Name}$$

The complete set of all resources in the cluster,  $R$ , is the disjoint union of these two sets:

$$R = R_{\text{ns}} \cup R_{\text{cl}}$$

Each resource  $r \in R$  is thus a tuple of its essential attributes. To refer to these resources, the language requires a clear and consistent identification method. This is achieved through a uniform syntax pattern that captures these attributes:

`type@namespace#name`

The application of this pattern is determined by the scoping discipline of the resource in question.

**Namespaced vs. Cluster-Scoped Kinds.** Kubernetes organizes resources into two fundamental categories based on their scope. The syntax of this language strictly reflects that distinction.

- *Namespaced kinds* are objects that exist only within a specific namespace, such as *Pod* or *Deployment*. For these resources, the namespace component of the identifier is *mandatory*.
- *Cluster-scoped kinds* are singleton, cluster-wide objects, such as *Node* or *Clusterrole*. For these, the namespace component is *omitted*, as it does not apply.

To facilitate broad queries, the syntax incorporates the wildcard character, `*`, which can be used in both the *type* and *namespace* positions. This capability facilitates selections not restricted to one single, known kind or a single namespace. For instance, `pods@*` is used to designate all Pods across every namespace, whereas `*@ns1` signifies all resources of any type situated exclusively within the *ns1* namespace. Combining them, `*@*`, selects all namespaced resources throughout the entire cluster, and a solitary `*` selects all cluster scoped resources.

The final section of the identifier, `#name`, is consistently optional and serves to dictate whether the selection should target a single, specific object or an entire collection. Although the *type* and *namespace* components fully support wildcards, the *name* component does not. Therefore, omitting the `#name` component entirely is the correct and sufficient syntax for selecting resources based on type and scope without needing to specify an individual name.

**Formal Denotation.** Each syntactic pattern refers to a precise semantic denotation. The formal definitions, including those for the type wildcard, are as follows:

$$\begin{aligned}
 type@ns\#name_R &= \{ r \mid kind(r) = type \wedge ns(r) = ns \wedge name(r) = name \} \\
 type@ns_R &= \{ r \mid kind(r) = type \wedge ns(r) = ns \} \\
 type@*_R &= \{ r \mid kind(r) = type \wedge kind(r) \in T_{ns} \} \\
 *@ns_R &= \{ r \mid ns(r) = ns \} \\
 type\#name_R &= \{ r \mid kind(r) = type \wedge name(r) = name \} \\
 type_R &= \{ r \mid kind(r) = type \} \\
 *@*\#name_R &= \{ r \mid name(r) = name \}
 \end{aligned}$$

**Example** The following examples illustrate the syntax in practice:

- `pods@ns1#nginx`: A point selection for a single namespaced object.
- `pods@ns1`: A group selection for all pods in a single namespace.
- `nodes`: A group selection for all cluster-scoped nodes.
- `pods@*`: A cross-namespace wildcard selection.
- `*@ns1`: A type wildcard selection for all resources in a single namespace.

**Note:** when using any wildcard with a specific name (e.g., `Role@*#editor`), is important to remember that the result may be a set of multiple objects. This is because namespaced resources are only required to be unique *within their own namespace*, and the same name can legitimately exist in several different namespaces.

## 6.1.2 Actions

*Actions* are the verbs within the RBAC model that specify the operations a subject may perform on the resource kinds defined in Section 6.1.1. They are the fundamental components used to express what a subject is authorized to do.

The language recognizes a fixed vocabulary of these verbs, which are formally defined as the set  $A$ :

$$A = \{get, list, watch, create, update, patch, delete, \dots\}$$

However, a crucial aspect of the RBAC model is that not all actions are meaningful for every kind of resource. To formally capture this constraint, the language defines a compatibility relation, denoted as `applies`, between the set of resource kinds,  $T$ , and the set of actions,  $A$ .

$$applies \subseteq T \times A$$

This relation holds true for a pair  $(t, a)$  if and only if the action  $a$  is an admissible and meaningful operation for the resource kind  $t$ . Any query that attempts to pair an action with an incompatible resource kind is considered semantically invalid.

### 6.1.3 Permissions

A *Permission* is the object that links actions to resources, forming the fundamental unit of authorization in RBAC. In this language, we define a single permission  $p$  as a pair, consisting of a set of actions,  $A_p$ , and a set of resource identifiers,  $R_p$ .

$$p = (A_p, R_p) \quad \text{where} \quad A_p \subseteq A \quad \text{and} \quad R_p \subseteq R$$

In the query language, these permission objects are created using a **permission** constructor. The general form is as follows:

$$\text{Permission}([ \text{actions} ], [ \text{resources} ])$$

Here, both *actions* and *resources* are themselves sets. For example, a single permission object could grant **get** and **list** actions on all **Pods** and **services** in a namespace. In practice, policy queries will operate on sets of these permission objects to evaluate a principal's access rights.

The `*` wildcard can be used within the permission constructor for both actions and resources to create broad permission statements.

- For the *actions* set, the `*` character represents all possible actions *applicable to the specified resource kind*. For instance, `Permission([*], [pods@ns1])` grants every valid action for a pod (such as **get**, **list**, **delete**, etc.) in the **ns1** namespace.
- For the *resources* set, the identifiers follow the comprehensive wildcard rules already detailed in Section 6.1.1, allowing for wildcards in the *type* and *namespace* fields (e.g., `Permission([get], [*@*])`).

### 6.1.4 Subject

*Subject* are the principals that receive RBAC bindings and on whose behalf permissions are evaluated. The domain is the disjoint union of users, service accounts, and groups:

$$S = S_{\text{user}} \cup S_{\text{sa}} \cup S_{\text{group}}$$

These are created with canonical constructors in the language:

$$\text{user}(\text{name}) \in S, \quad \text{sa}(\text{name}@ns) \in S, \quad \text{group}(\text{gname}) \in S$$

Users and groups are global identifiers, whereas service accounts are namespaced identities and therefore require a namespace by construction. This distinction is reflected in well-formedness constraints for subject terms.

The `*` wildcard can be used to select groups of subjects by type rather than by a specific name or namespace. This is particularly useful for type-driven queries.

- `user(*)`: Selects the set of all users.
- `group(*)`: Selects the set of all groups.

- `sa(*@ns1)`: Selects all service accounts within the `ns1` namespace.
- `sa(deployer@*)`: Selects all service accounts named `deployer` across all namespaces.
- `sa(*@*)`: Selects the set of all service accounts across all namespaces.

It is important to note that these three constructors all produce elements of the single, unified *Subject* domain,  $S$ . A key feature of this design is that these types are interchangeable. Consequently, any language construct that operates on a set of subjects can accept a mixed set containing any combination of users, groups, and service accounts. For example, the set `user(dev1)`, `group(admins)`, `sa(robot@ns1)` represents a valid, heterogeneous sub-collection of *Subject*.

### 6.1.5 Role

*Role* are named policy objects that encapsulate permissions. The language treats namespaced roles and cluster roles as elements of a single domain with two disjoint constructors. Let *Role* denote the set of roles, partitioned into namespaced and cluster-scoped roles:

$$\text{Role} = \text{Role}_{\text{ns}} \cup \text{Role}_{\text{cl}}.$$

The canonical surface constructors are:

$$\text{role}(\text{name}@ns) \in \text{Role}_{\text{ns}} \quad \text{and} \quad \text{clusterrole}(\text{name}) \in \text{Role}_{\text{cl}}.$$

Namespaced roles require a namespace by construction; cluster roles are cluster-scoped and therefore carry no namespace.

The `*` wildcard can be used to select groups of roles by type rather than by a specific name or namespace.

- `role(*@ns1)`: Selects all roles within the `ns1` namespace.
- `role(editor@*)`: Selects all roles named `editor` across all namespaces.
- `role(*@*)`: Selects the set of all namespaced roles across all namespaces.
- `clusterrole(*)`: Selects the set of all cluster-scoped roles.

Following the same principle as the *Subject* domain, the language treats namespaced and cluster roles as interchangeable elements of a single, unified *Role* domain. This allows any language construct that operates on a set of roles to accept a heterogeneous collection, such as `role(editor@ns1)`, `clusterrole(admin)`.

## 6.2 NetworkPolicy Value Domains

This section defines the value domains used by the language for NetworkPolicy reasoning. We introduce the sets for *protocols* and *ports*, then define *network endpoints* in a way that reuses the resource notation from 6.1.1. No operators or query forms are introduced here; we only specify the underlying domains.

### 6.2.1 Protocols

Let `Proto` denote the transport protocols recognized by Kubernetes NetworkPolicy, extended to include a wildcard.

$$\text{Proto} = \{\text{TCP}, \text{UDP}, \text{SCTP}, *\}.$$

The wildcard symbol `*` is included to represent any of the supported transport protocols.

### 6.2.2 Ports

Let `PortNum` =  $\{1, \dots, 65535\}$  denote the set of valid numeric transport ports. To allow for policies that match any port, the domain of ports used by the language, `Port`, is the union of this set and the wildcard symbol:

$$\text{Port} = \text{PortNum} \cup \{*\}.$$

### 6.2.3 Network Endpoints

NetworkPolicy endpoints identify sets of pods that act as sources or targets. We reuse the resource notation of 6.1.1 and specialise it to pod selections.

An endpoint term has the form

$$\text{Pod@namespace\#refinement}$$

Namespace field can be a specific one or the wildcard character `*` to represent all namespaces. The `\#refinement` is optional and, when present, is a name or a label:

$$\text{name} \quad | \quad \text{key=value}.$$

The grammar makes the two refinements mutually exclusive. The wildcard is only applicable to the namespace; selecting all pods within a given scope is achieved by omitting the `\#refinement`.

**Formal Denotation.** Each syntactic endpoint pattern is mapped to its semantic denotation as follows, fixing the kind to `pods`:

$$\begin{aligned} \text{Pod@ns}_E &= \{ r \mid \text{kind}(r) = \text{Pod} \wedge \text{ns}(r) = ns \} \\ \text{Pod@ns\#name}_E &= \{ r \mid \text{kind}(r) = \text{Pod} \wedge \text{ns}(r) = ns \wedge \text{name}(r) = name \} \\ \text{Pod@ns\#key=value}_E &= \{ r \mid \text{kind}(r) = \text{Pod} \wedge \text{ns}(r) = ns \wedge \text{label}(r, key) = value \} \\ \text{Pod@*}_E &= \{ r \mid \text{kind}(r) = \text{Pod} \} \\ \text{Pod@*\#key=value}_E &= \{ r \mid \text{kind}(r) = \text{Pod} \wedge \text{label}(r, key) = value \} \end{aligned}$$

## Examples

- `Pod@team-a`  $\Rightarrow$  all pods in namespace `team-a`.
- `Pod@team-a#nginx`  $\Rightarrow$  the pod named `nginx` in `team-a`, if it exists.
- `Pod@team-a#app=web`  $\Rightarrow$  all pods in `team-a` with label `app=web`.
- `Pod@*`  $\Rightarrow$  all pods across all namespaces in the cluster.
- `Pod@*#app=db`  $\Rightarrow$  all pods with the label `app=db` in any namespace.

## 6.3 Predicate Queries

The most fundamental operation in the language is the **predicate query**. These are boolean expressions, returning **True** or **False**, designed to answer direct yes/no questions about the state of cluster policies. Their primary purpose is to verify whether there is a specific relationship between a set of elements and a set of specified properties. This section defines the syntax for these queries, their validation rules, and their formal semantics.

### 6.3.1 Query Syntax and Semantics

The syntax for a predicate query is:

$$[ \textit{root} ] \text{ has } ( [ \textit{predicate} ] )$$

The **root set** identifies the subject of the query. It can be a set of *subjects*, a set of *roles*, or a set of *network endpoints*. The language treats a single element as a singleton set, unifying the handling of individual and collective queries.

The **predicate set** specifies the properties to be verified against the root; this entire “has ( [predicate set] )” clause is referred to as the **predicate form**. As will be shown, this form is a fundamental building block reused in more complex query structures. The predicate set can be made up with one of these four types:

- **Permission:** A set of Permission pairs, where each permission is a tuple (action, resource).
- **Connection:** A set of Connection tuples, where each connection is defined by (Network Endpoint Destination, Port, Protocol, mode), with *mode*  $\in$  {ingress, egress}.
- **Role:** A set of roles, as defined in Section 6.1.5.
- **Subject:** A set of subjects, as defined in Section 6.1.4.

Although the grammar of the language allows for numerous combinations of root and predicate types, not all of these pairings are semantically meaningful. During the design of this language, a careful analysis was conducted to distinguish between logically coherent queries and those that are conceptually invalid. This involved excluding combinations such as cross-domain queries (e.g., asking if a network endpoint has an RBAC permission) and reflexive queries (e.g., checking if a subject has another subject).

The table below summarizes the outcome of this semantic validation, where a checkmark (✓) indicates a valid pairing between a root type and a predicate parameter. Crucially, these semantic rules are embedded directly into the language’s formal syntax; only the combinations shown as valid are permitted.

Root / Parameter	Role	Subject	NE
Permission	✓	✓	
Connection			✓
Role		✓	
Subject	✓		

## Evaluation Semantics and Quantifiers

Given that both the root and the predicate are defined as sets, a question arises regarding the default evaluation logic: for a query to be true, must all elements of the sets satisfy the condition, or is it sufficient for just one? The language answers this by defining a default behavior and providing a modifier to express alternative semantics.

**Default Behavior: Universal Quantification.** By default, a predicate query is interpreted with maximum strictness. An expression **R has P** evaluates to **true** if and only if *every* element in the root set  $R$  satisfies *every* predicate in the set  $P$ . This is equivalent to a series of logical AND operations across all combinations. Formally, this default behavior is a universal quantification ( $\forall$ ) over both sets:

$$\forall r \in R, \forall p \in P : \text{has}(r, p)$$

**Modified Behavior: Existential Quantification.** This strict universal logic can be relaxed using the **+** modifier. This operator changes the quantifier for a set from universal (*all*,  $\forall$ ) to existential (*at least one*,  $\exists$ ), which is equivalent to using a logical OR. The **+** can be applied compositionally to the root set, the predicate set, or both, to fine-tune the query’s logic.

The modified semantics are as follows:

**+ R has P** *At least one* root must satisfy *all* predicates.

$$\exists r \in R, \forall p \in P : \text{has}(r, p)$$

**R has + P** *Every* root must satisfy *at least one* predicate.

$$\forall r \in R, \exists p \in P : \text{has}(r, p)$$

**+ R has + P** *At least one* root must satisfy *at least one* predicate.

$$\exists r \in R, \exists p \in P : \text{has}(r, p)$$

### 6.3.2 Examples

1. **Query:** Check if the service account `deployer` in namespace `team-a` has the `Permission` to create `Pod` in the same namespace.

**Syntax:**

```
[sa(deployer@team-a)] has ( [Permission([create],
                             [Pod@team-a])] )
```

This query involves a singleton root set and a singleton predicate set.

2. **Query:** Verify that the `editor` role in namespace `dev` grants permissions to both update `Deployment` and get `Pod`.

**Syntax:**

```
[role(editor@dev)] has (
    [Permission([update], [Deployment@dev]),
     Permission([get], [Pod@dev])] )
```

This query checks if one root element satisfies all predicates in a set of two.

3. **Query:** Verify that all pods with the label `app=frontend` in the `web` namespace are allowed to make outgoing (egress) connections to any pod with the label `app=backend` on TCP port 8080.

**Syntax:**

```
[pods@web#app=frontend] has (
    [Connection(Pod@web#app=backend, 8080, TCP, egress)]
)
```

This query evaluates a property over a set of root network endpoints. It will return **True** only if every frontend `Pod` has the specified egress permission.

4. **Query:** Illustrate the Cartesian product evaluation by checking if both `user1` and `user2` have two distinct permissions.

```
[ user('user1'), user('user2') ] has (
    [ Permission([get], [Pod@ns1]),
      Permission([create], [Deployment@ns1]) ]
)
```

**Evaluation Detail** This query demonstrates the evaluation semantics over sets. The query engine computes the Cartesian product of the root set (`{user1, user2}`) and the predicate set, resulting in  $2 \times 2 = 4$  distinct checks:

- Does `user1` have permission to `get Pod@ns1`?
- Does `user1` have permission to `create Deployment@ns1`?
- Does `user2` have permission to `get Pod@ns1`?
- Does `user2` have permission to `create Deployment@ns1`?

The entire query returns **True** if and only if all four of these individual checks succeed.

### 6.3.3 Predicate Query Negation

To verify the *complete absence* of a relationship, the language provides the ‘not’ operator. This form evaluates to True if and only if *all* elements in the root set fail to satisfy *all* predicates in the predicate set. In other words, for the query to succeed, every possible combination of a root element and a predicate must be false. Formally, the query `R not has P` is true if:

$$\forall r \in R, \forall p \in P : \neg \text{has}(r, p)$$

The existential modifier `+` can also be applied to a `not has` expression; the evaluation logic remains consistent with what has been explained earlier.

**Example.** A common use case is to enforce strict security boundaries.

**Query:** Verify that the user `intern-1` has absolutely no permissions related to deleting cluster-level node objects.

**Syntax:**

```
[user(intern-1)] not has ([Permission([delete], [Node])])
```

This query succeeds only if the `intern-1` user lacks the specified `delete` permission on `Node`, confirming a strict compliance with a least-privilege policy.

### 6.3.4 Combining Predicate Queries

Individual predicate queries (both affirmative and negated) can be composed into larger, more expressive statements using the logical operators `and` and `or`. This compositional capability is what allows the language to span multiple policy domains within a single, coherent query.

The general syntax is:

$$\begin{aligned} & \textit{predicate\_query\_1} \text{ and } \textit{predicate\_query\_2} \\ & \textit{predicate\_query\_1} \text{ or } \textit{predicate\_query\_2} \end{aligned}$$

The semantics are standard: for `and`, both sub-queries must be true; for `or`, at least one sub-query must be true.

## 6.4 Function Queries

To support complex auditing and discovery tasks, the language provides **Function queries**. In contrast to their boolean counterparts, these queries do not return a simple validation result but instead yield a *set of values*. This allows an operator to list all resources, permissions, or other objects that satisfy a specific set of criteria.

The general structure reuses the predicate form as its filtering mechanism:

$$[ \textit{root} ] \textit{list} ( \textit{Selector}, \textit{Predicate Form} )$$

### 6.4.1 Query Syntax and Semantics

**Root Set** The initial set of objects to be evaluated. As in predicate queries, the root can be a set of *subjects*, a set of *roles*, or a set of *network endpoints*.

**Selector** Specifies what kind of data the query should return. We distinguish between two types of selectors:

- *Standard Selectors: Actions, Subjects, Roles, Resources, Permissions, Connections.* These selectors extract data associated with the root elements that satisfy the predicate form.
- **Special Selector: Select.** This selector is unique: it does not extract associated data, but instead returns the subset of the original root elements that satisfy the predicate form.

**Predicate Form** The filtering condition for the query. This component is an instance of the **predicate form** (`has(...)`), as defined in Section 6.3. Specifically, it can take one of the following forms:

- `has([Permission])`
- `has([Connection])`
- `has([Subject])`
- `has([Role])`

### Formal Evaluation Semantics

The behavior of a Function query can be formally understood by decomposing its execution into a two-phase process. This model clarifies the relationship between *Function* and *Predicate* queries, showing the former as an extension of the latter. The conceptual model is that a query of the form `R list(S, P)` first finds the subset of `R` that satisfies the predicate `P`, and subsequently applies the selector `S` to that subset to aggregate a final result.

The first phase is a **filtering** operation, which is identical in function to a predicate query. It iterates through the root set `R` to produce a subset, denoted as  $R_{\text{match}}$ , containing only those elements that satisfy the predicate `P`. We can define this subset formally as follows:

$$R_{\text{match}} = \{r \in R \mid \mathbf{r} \text{ has } P \text{ evaluates to true}\}$$

The condition  $\mathbf{r} \text{ has } P$  represents the boolean evaluation of the predicate query as defined in Section 6.3. If the resulting  $R_{\text{match}}$  set is empty, the Function query’s execution concludes, returning an empty set.

Once the  $R_{\text{match}}$  subset is identified, the second phase, **aggregation**, begins. This phase operates exclusively on the elements within  $R_{\text{match}}$ . For each element  $r \in R_{\text{match}}$ , the query extracts a set of values  $V_r$  as specified by the selector. The final result is determined by the method used to combine these individual value sets ( $V_{r_1}, V_{r_2}, \dots$ ).

By default, the aggregation logic performs a mathematical **intersection** ( $\cap$ ). This corresponds to a logical AND, meaning the final result contains only elements that are present in the value sets of *every* element in  $R_{\text{match}}$ . This is useful for identifying strictly shared properties. The final set of results,  $Q$ , is defined as:

$$Q = \bigcap_{r \in R_{\text{match}}} V_r$$

Alternatively, when the  $*$  modifier is prepended to the root set, the aggregation logic is changed to a mathematical **union** ( $\cup$ ). This corresponds to a logical OR, where the final result contains any element present in the value set of *at least one* element in  $R_{\text{match}}$ . The final result set is then defined as:

$$Q = \bigcup_{r \in R_{\text{match}}} V_r$$

The **Select** selector constitutes a special case that alters this two-phase model. Its function is solely to execute the filtering phase and return the resulting  $R_{\text{match}}$  set directly. As it does not perform the second phase of value extraction and aggregation, the  $+$  modifier is not applicable when **Select** is used.

It is important to note that the predicate form used in this filtering phase is fully featured. This means it can include the existential modifier  $+$  or be a **not has** expression, following the semantics defined in the section on Predicate Queries. For example, if the predicate form is **has(+ P)**, a root element  $\mathbf{r}$  is included in the  $R_{\text{match}}$  set if it satisfies *at least one* of the conditions in  $P$ . Likewise, if the form is **not has(P)**, the filtering will select only those root elements for which the negation condition holds true.

## Semantic Validation

Similar to predicate queries, Function queries are subject to a semantic validation phase. In the design of this language, this validation ensures that only combinations that represent a logically sound question are permitted. The validation rules are specific to the type of root set. We begin by analyzing the rules for a *Subject* root.

**Validation for a Subject Root** When the root of a Function query is a set of subjects (e.g., `user(dev1)`, `group(admins)`), not all selectors are compatible with all predicates. The permitted combinations are outlined in the table below.

Selector / Predicate	Actions	Roles	Subjects	Resources	Select	Connections	Permission
<code>has([permission])</code>	✓	✓		✓	✓		✓
<code>has([connection])</code>							
<code>has([Subject])</code>							
<code>has([Role])</code>	✓	✓		✓	✓		✓

Table 6.1. Valid Selector and Predicate combinations for a Subject root.

The logic underpinning these validation rules rests upon a set of core principles established to safeguard the model’s semantic integrity. The foremost principle ensures that all combinations which are either *cross domain* or conceptually meaningless are explicitly forbidden. For example, a query starting with a *Subject* root cannot be evaluated with a `has([connection])` predicate, since the model dictates that subjects do not possess network connections. Similarly, the `has([Subject])` predicate is invalid when applied to a *Subject* root. This is because the underlying policy structure defines a subject’s relationships strictly by the **roles** it possesses or the specific **permissions** it holds, not through relationships to other subjects. Since the semantic relationship of one subject *having* another is undefined, any such query is logically rejected as being non-sensical.

Secondly, combinations where the query selector specifies an identical type to the root are deliberately prohibited. For instance, the query `Subject list(Subjects, ...)` is invalid. This prohibition exists because the action of filtering an initial root set based on a predicate is already and explicitly handled by the `Select` selector. This design choice imposes a clearer and more direct query structure: when the intent is to filter the original set of subjects, `Select` remains the designated and unambiguous operation for that task.

**Validation for a Role Root** Next, we consider the validation rules when the root of the query is a set of roles (e.g., `Role@ns1`, `clusterroles`). The same principles of semantic integrity apply, resulting in a distinct set of valid combinations as shown in the table below.

Selector / Predicate	Actions	Roles	Subjects	Resources	Select	Connections	Permission
<code>has([permission])</code>	✓		✓	✓	✓		✓
<code>has([connection])</code>							
<code>has([Subject])</code>	✓		✓	✓	✓		✓
<code>has([Role])</code>							

Table 6.2. Valid Selector and Predicate combinations for a Role root.

The logic behind these rules remains consistent with the previous case. Cross-domain queries, such as evaluating a *Role* against a `has([connection])` predicate, are disallowed as they have no semantic meaning. Similarly, a predicate of `has([Role])` is not permitted for a *Role* root, as the language model does not define a direct relationship where one role *has* another. A role's primary relationships are with the **permissions** it contains and the **subjects** to which it is bound.

Finally, the `Roles` selector is invalid for a *Role* root for the same reason outlined before: `Select` is the designated, unambiguous mechanism for filtering and returning elements from the original root set.

**Validation for a Network Endpoint Root** Finally, we analyze the validation rules for a Function query where the root is a set of network endpoints. As these entities exist purely within the network policy domain, the set of valid combinations is highly constrained.

Selector / Predicate	Actions	Roles	Subjects	Resources	Select	Connections	Permission
<code>has([permission])</code>							
<code>has([connection])</code>					✓	✓	
<code>has([Subject])</code>							
<code>has([Role])</code>							

Table 6.3. Valid Selector and Predicate combinations for a Network Endpoint root.

As the table illustrates, only two combinations are allowed, as the majority of potential queries are rejected due to invalid cross-domain semantics. For instance, a `network_endpoint` has no relationship with RBAC concepts such as `Roles`, `Subjects`, or `Permissions`.

This highlights a key principle: a query is only valid if its predicate is semantically meaningful. A query like `subject list(Subjects, has([Subject]))` is invalid because the predicate `has([Subject])` does not represent a meaningful property to verify on a `subject` root. This makes the entire expression semantically unsound from the start, in contrast to valid queries where the predicate is a meaningful operation within the data model.

## 6.4.2 Examples

**Example 1: Finding Shared Roles (Intersection)** **Goal:** Find all the roles that are shared by both `user1` and `user2`.

```
[user(user1), user(user2)] list (
  Roles,
  has(+ [role(*)])
)
```

**Explanation:** The query finds the roles for `user1` and the roles for `user2`, and then returns the **intersection** of these two sets. The predicate `has(+`

`role(*)`) simply ensures that we only consider users who have at least one role.

**Example 2: Collecting All Roles (Union)** **Goal:** Collect a single list of all unique roles assigned to either `user1` or `user2`.

```
+ [(user(user1), user(user2))] list (
  Roles,
  has(+ [role(*)])
)
```

**Explanation:** The `+` before the root set switches the logic. The query finds the roles for each user and returns the **union** of the sets.

**Example 3: Filtering with Select** **Goal:** Find all ‘ClusterRoles’ that grant permission to delete Node.

```
+ [clusterrole(*)] list (
  Select,
  has(Permission(delete, Node))
)
```

**Explanation:** This query inspects all `Clusterrole`. The `Select` selector returns the `clusterrole` objects themselves if they satisfy the predicate of having the `delete Node` permission.

### 6.4.3 Combining Function Queries

Beyond the logical `and` and `or` operators used to combine predicates, the language provides set-based operators to compare and manipulate the results of two Function queries. The `in` and `not in` operators are used to generate subsets, returning a set of elements that satisfy the relationship.

The syntax for a set-based query is as follows:

```
Function_query_A in Function_query_B
Function_query_A not in Function_query_B
```

A fundamental constraint of these operators is the enforcement of strict *type safety*. A set query is deemed semantically valid only if both operands resolve to sets of a homogeneous type and their list selectors are identical. Any attempt to compare sets of different types, for example, a set of `Roles` against a set of `Subjects`, is invalid.

**The in Operator (Intersection)** The `in` operator returns the elements from the result set of `Function_query_A` that are also present in the result set of `Function_query_B`. The result is therefore the intersection of the two sets.

Formally, if  $Q_A$  and  $Q_B$  are the result sets of the two queries, the expression returns:

$$Q_A \cap Q_B$$

**The ‘not in’ Operator (Difference)** The `not in` operator returns the elements from the result set of `Function_query_A` that are not present in the result set of `Function_query_B`. The result is the difference of the two sets.

Formally, the expression returns:

$$Q_A \setminus Q_B$$

## Example

A primary use case for set queries is to compare related data sets, such as the permissions between hierarchical roles. For example, a security policy might want to precisely identify which permissions of a custom role exceed those of a built-in role.

**Goal:** Identify which permissions of the `developer` role are not contained within the `cluster-admin` role.

**Analysis:** This can be achieved by generating the permission set for each role and using the `not in` operator to return the elements that are present in the first set but not in the second.

**Syntax:**

```
[role(developer@default)] list (  
  Permissions,  
  has(+ [Permission(*,*)])  
)  
not in  
[clusterrole(cluster-admin)] list (  
  Permission,  
  has(+ [Permission(*,*)])  
)
```

This query first executes the two Function queries to produce two sets of permissions. It then returns the set of permissions that are in the `developer` role’s set but not in the `cluster-admin` role’s set, providing a precise analysis of misaligned permissions.

### 6.4.4 Cardinality Queries as Function-Query Extensions

This section introduces **cardinality queries**, a specialized extension of function queries designed to evaluate the size of a resulting set against a numeric threshold. Although these queries return a boolean value and could thus be classified as *predicate queries*, they adopt the function-query syntax to maintain semantic clarity and consistency.

**Syntax.** The concise syntax for a cardinality query is as follows:

```
[ root ] list( Selector, Predicate Form ) IntegerOp Integer
```

where the components are defined as:

- **IntegerOp:** A comparison operator from the set {<, >, =, <=, >=}.
- **Integer:** A non-negative integer representing the threshold for comparison.
- The **selector** and **predicate form** adhere to the same semantic validation rules established for standard function queries.

The expression evaluates to **true** if and only if the number of elements returned by the `list()` operation satisfies the specified integer comparison; otherwise, it evaluates to **false**.

Cardinality queries are subject to specific compositional constraints:

- **No Set-Containment Composition.** As cardinality queries resolve to a boolean value rather than a set, they *cannot* be combined with other function or cardinality queries using the `in` or `not in` operators.
- **Boolean Composition.** They can be integrated with standard predicate queries using boolean connectives (e.g., `and`, `or`). This allows for the construction of complex logical expressions, such as guardrails or budget-style constraints.

**Interpretation.** The query structure is interpreted as follows:

- **Root:** The optional `[root]` element defines the initial scope for the query (e.g., the principal or role being evaluated).
- **list():** This function produces a set based on the provided *selector* and *predicate form*. The subsequent comparison operator is then applied to the *cardinality* (i.e., the size) of this resulting set.

Although, the following design constraints must be taken into account for a proper semantic interpretation:

- Cardinality queries are boolean by design, which allows for their natural integration with the logic of predicate queries, while the `list()` syntax provides a clear expression of intent.
- All typing, domain, and validation restrictions applicable to the selector and predicate form in function queries are inherited by cardinality queries.

## 6.5 Advanced Queries: Discovery and Type-Driven Auditing

While standard queries are useful for verifying properties of known entities, a more advanced class of queries is needed for discovery and broad auditing. This is achieved by combining the universal wildcard (\*) with the existential modifier + in the root of an expression.

This + `type(*)` pattern enables a shift from *entity-driven* queries, which operate on known objects, to *type-driven* or *property-driven* queries, where the focus is on the property being audited rather than a specific, known subject. The key advantage is the ability to audit the system with reduced knowledge; an operator does not need to know the names of specific principals beforehand. The pattern's behavior depends on the query type:

- In a **predicate query**, the pattern performs an existence check. The expression + `type(*) has P` answers the boolean question: “Does at least one object of this type satisfy the predicate?”
- In a **Function query**, the pattern becomes a powerful discovery tool. It allows an operator to list the attributes specified by the *Selector* for every element of the root *type* that satisfies the *Predicate*. The final result is the **union** of the data extracted from all matching elements, providing a complete list of all entities that possess a certain property. As detailed in Section 6.4, the **Select** selector is an exception to this pattern; since its purpose is to filter rather than aggregate extracted data, it is not used with the + modifier on the root.

**Predicate query Example: Verifying Existence.** A fundamental security audit is to discover if any user has been granted the highest level of privilege, without needing a list of all users to check.

**Goal:** Discover whether there is at least one user in the entire system who is bound to the `cluster-admin` role.

**Syntax:**

```
+ [user(*)] has ([clusterrole(cluster-admin)])
```

**Explanation:** This query inspects the set of *all* users and returns `true` if it finds even one user who satisfies the predicate of having the `cluster-admin` role.

**Function query Example: Discovering All Roles in Use.** This pattern can be used to discover the complete set of privileges assigned to a certain class of principals.

**Goal:** Discover the complete and unique set of all roles that are assigned to *any* service account across the entire cluster.

**Syntax:**

```
+ [sa(*@*)] list (
  Roles,
  has(+ [role(*)]) )
```

**Explanation:** This query inspects every service account in all namespaces. For each `sa` that satisfies the predicate (i.e., has at least one role), it extracts the set of roles bound to it. The `+` modifier on the root `sa(*@*)` then ensures that all of these individual sets of roles are combined into a single, comprehensive set via a *union*. The result is a complete list of every unique role used by service accounts in the cluster.

## 6.6 Example Table

The following table (Section 4.1) presents the various query categories supported by the system. For each one, a two-part example is provided: first, a theoretical question expressed in natural language to represent the operator’s intent, followed by its formal translation into the defined language to show its concrete syntactical implementation.

Table 6.4: Operator-friendly query families with concise DSL representation and examples.

Query Name	Query	Example
Can I? (Authorization)	<code>[sa(sa-ci@builds)] has ([Permission([update], [Deployment@build#frontend])])</code>	can sa-ci@builds update Deployment@builds #frontend?
Who is bound to what?	<code>[user(alice@corp)] has ([clusterrole(view)])</code>	Is user alice@corp bound to clusterrole/view?
Explore & List (RBAC)	<code>[user(*)] list( Select, has([Permission([create], [Pod@*])]))</code>	List subjects that can create Pod.
Permission Delta	<code>[user(user1)] list(Permissions, has([Permission([*, [*])])) not in [user(user2)] list(Permissions, has([Permission([*, [*])]))</code>	What can user1 do that user2 cannot?
Cardinality Constraint	<code>+ [ user(*), sa(*@*)] list( Select, has( [Permission( [update], [Deployment@*] )] ) ) &lt;= 3</code>	$\leq 3$ subjects may update deployments.

Query Name	Query (DSL)	Example
<b>Allowlist Only</b>	<code>[user(cluster-admin)] has( [Permission( [delete], [Namespace])] ) and +[user(*), sa(*@*)] list( Select, [Permission(delete, Namespace)] ) = 1</code>	Only <code>cluster-admin</code> can delete namespaces.
<b>Can it talk? (Reachability)</b>	<code>[Pod@app#frontend] has ([Connection(Pod@app#backend, 5000, TCP, egress)])</code>	can <code>Pod@app#frontend</code> communicate to <code>Pod@app#backend</code> on <code>TCP/5000</code> ?
<b>Namespace Isolation</b>	<code>[pod@*#app=demo] not has([connection(*,*,*,ingress), Connection(*,*,*,egress)])</code>	No ingress and egress connection for pod with label <code>app=demo</code> .
<b>External Exposure Check</b>	<code>+ [Pod@*] not has([Connection(0.0.0.0/0, 8080, TCP, egress)])</code>	No Pods have egress to <code>0.0.0.0/0</code> on <code>TCP/8080</code> .
<b>Directionality</b>	<code>+ [Pod@*#np-frontend] not has([Connection(*, *, *, egress)])</code>	Verify that Pod named <code>np-frontend</code> is ingress-only.
<b>Explore &amp; List (Network)</b>	<code>[Pod@*#app=frontend] list(Connections, +has([Connection(*, *, *, ingress)]))</code>	List all ingress connections from sources with label <code>app=frontend</code> .
<b>Dangerous Combinations</b>	<code>( +[sa(*@*)] not has([Permission([update], [Deployment@prod])]) ) and ( +[Pod@prod] not has([Connection(0.0.0.0/0, *, *, egress)]) )</code>	No serviceaccount may update Deployment in <code>prod</code> <i>and</i> Pods in the same namespace do not have egress to internet.
<b>Zombie Roles</b>	<code>[role(*)] list( Select, not has(+[user(*)]) and not has(+[sa(*@*)]) and not has(+[group(*)]) )</code>	List roles left over from past migrations.

# Chapter 7

## Proof of Concept: Implementation and Validation

This chapter operationalizes the architectural principles introduced in Chapter 5 by presenting a working *proof of concept* (PoC) that both implements the proposed pipeline and evaluates its effectiveness on representative scenarios. The aim is twofold. First, to show that the unified data model, the typed graph, and the logic-based reasoning core can be realized in a cohesive system with clear interfaces and predictable behavior. Second, to demonstrate that this realization provides *useful and verifiable answers* to operator-centric security questions, thereby reducing cognitive load and shortening verification cycles in practice.

The chapter is organized into two main parts. The first, *Implementation*, describes how the architectural components are realized end-to-end. It details the ingestion pipeline for producing the cluster snapshot, the construction of the typed directed graph including the resource backbone, the authorization overlay, the connectivity overlay, and the translator that emits a compact Prolog theory together with the query goals. Practical aspects such as tooling choices, data structures, encoding decisions, and observability hooks are discussed to clarify the trade-offs taken in the design.

The second section, titled *Validation*, provides a practical demonstration of the Proof of Concept (PoC) and details the resulting outcomes. A comprehensive evaluation protocol is established to accurately simulate the needs of cluster operators: specific scenarios are engineered to test queries spanning multiple domains, to combine both authorization and network connectivity constraints, and to identify complex edge cases, such as interactions involving the scope of *ClusterRole* objects and namespaced bindings.

**Structure of the Chapter** Section 7.1 (*Implementation*) details the system realization, interfaces, and internal representations. Section 7.2 (*Validation*) introduces the demo environment, the evaluation protocol and metrics, and then presents and discusses the results. Together, they provide the empirical grounding that connects the architecture of Chapter 5 to concrete,

verifiable outcomes, and they identify the most promising directions for extension and industrialization.

## Experimental Environment and Tooling Versions

To ensure reproducibility and contextualize performance results, Table 7.1 reports the software stack and host platform used during the PoC runs.

Table 7.1. Environment, tools, and versions used in the PoC

Component	Version / Details
Kubernetes client ( <code>kubectl</code> )	v1.22.4
Python	3.12.0
SWI-Prolog	9.2.9 (x64-win64)
CPU	Intel Core i7-1195G7 (11 <sup>th</sup> gen., 4C/8T)
RAM	16 GB
Operating system	Windows 11 24H2

## 7.1 Implementation

This section details how the architecture is realized in the Proof of Concept. The description proceeds in the same order as the data flow: scope and simplifications, snapshot acquisition in the cluster, parsing and graph construction in Python, RBAC integration including namespaced bindings to cluster roles and cluster role aggregation, NetworkPolicy normalization, and finally the Prolog bridge and rule base that perform the verification.

### 7.1.1 Simplifications Adopted in the PoC and Their Impact on Validity

To keep the PoC focused, and aligned with the thesis objectives, a small number of simplifications were adopted. These choices narrow the domain without altering the core mechanisms of the system or the accuracy of the results.

The API coverage is deliberately restricted to four Kubernetes groups, namely `core/v1`, `apps/v1`, `rbac.authorization.k8s.io/v1`, and `networking.k8s.io/v1`. This selection is sufficient to enumerate resources and labels, capture the main deployment primitives, model the RBAC authorization plane, and source network connectivity policies. In practice, it provides everything required to construct the typed graph, attach permissions to individual instances or to classes of resources, and reason about connectivity at the level of admitted flows. Broader domains such as admission control, Pod Security, and custom resources are not necessary to demonstrate the verification pipeline and they remain natural extensions of the model.

Connectivity is handled through an explicit normalization step that abstracts `NetworkPolicy` specifications into admitted flows between classes of endpoints, annotated with transport attributes. Endpoint selection is derived from `podSelector` using exact `matchLabels`.

Features such as `matchExpressions`, `namespaceSelector`, and composite selector combinations are not employed in the present iteration, yet they can be incorporated with minimal changes to the abstraction. Because the verification engine operates over the normalized extent of selected Pod sets and the rules that admit traffic, it only requires the snapshot to include the induced admitted flows and the selective information used to compute them, not the full breadth of policy features.

A final simplification was made in the query system. Not all possible queries were implemented; rather, a representative subset was chosen sufficient to demonstrate the core capabilities introduced thus far. Furthermore, the parser that translates queries from the defined high-level language into Prolog requests was not implemented. Instead, this translation was performed manually for the selected queries. This approach does not compromise the validity of the results, as the core verification engine is still tested with well-formed logical requests. It simply streamlines the proof of concept by separating the core verification logic from the significant engineering effort of building a query compiler, keeping the implementation focused and lightweight.

These simplifications do not undermine the validity of the proof of Concept. They prioritize soundness over completeness, since every positive answer is justified by explicit facts in the snapshot and by derivations in the graph, with no reliance on implicit assumptions. They preserve representative expressivity for the central operational questions, including the composition of permissions with protocol and port constrained connectivity. They also maintain monotonic extensibility. Widening the API surface, enabling richer selectors, or implementing a full query parser amounts to adding well-typed node and edge kinds and a small number of local rules, without changing the query style or invalidating existing results. The narrower perimeter keeps the proof-of-concept compact and transparent while remaining sufficient to demonstrate the value of the architecture and ready to grow incrementally.

### **7.1.2 Cluster Information Acquisition**

The first step of the implementation produces a coherent snapshot of the cluster that serves as the single source of truth for graph construction and query evaluation. The collection layer is read-only, idempotent, and portable. It relies on `kubect1` as a stable interface to the API server, with lightweight shell and small `python3` helpers where JSON transformation or discovery-driven iteration is convenient. The scripts operate on the bounded API surface already introduced, and they emit plain, line-oriented text files that the parser can ingest deterministically.

For the resource inventory, the scripts enumerate kinds and possible sub-resources via API discovery and then list actual instances. Each object

is projected into a single record containing its canonical Kubernetes path, the exact set of admissible verbs as reported by discovery, and its labels (when present). Typical paths include `/api/v1/namespaces/app-ns/pods/app-pod-123` for core resources and `/apis/apps/v1/namespaces/webapp/deployments/frontend/scale` when a subresource is present. Collecting the full set of admissible verbs for each resource is essential to enable the ‘\*’ wildcard in queries, where ‘\*’ denotes “all verbs admissible for that resource.” Labels are preserved specifically to support NetworkPolicy selector matching in later stages. The resulting inventory is then written out in a line-oriented, deterministic order.

For authorization, roles and bindings are exported *as raw YAML manifests* using `kubectl`. This procedure yields two separate, multi document collections: one encompassing all `ClusterRole` and `Role` definitions, and a second containing all `ClusterRoleBinding` and `RoleBinding` definitions, all copied precisely from the API server’s output. A subsequent, minor flattening operation then converts each document into a singular, line oriented record. These records serve as the core inputs for constructing the authorization overlay, thereby ensuring a distinct separation between the initial declared policy data and all subsequent derivations.

For connectivity, we likewise use `kubectl` to export all `NetworkPolicy` resources *as raw YAML* into a single multi-document manifest. A minimal post-processing pass then emits a line-oriented export, containing the normalized records required by the parser.

These design choices reflect the architectural separation between collection and interpretation. Inputs stay close to the declarations the cluster exposes, transformations are minimal and repeatable, and the products are stable text files with explicit semantics. The result is a snapshot that is faithful to the observed state, a graph that retains the granularity needed for cross-domain verification, and a pipeline that remains easy to extend by adding further API groups or richer selectors without changing the nature of the flow.

### 7.1.3 Unified Graph Construction

Graph construction follows the scheme presented in the architecture chapter. First, the structural *backbone* is materialized through containment relations. Then, two policy overlays, namely authorization and connectivity, are added while keeping types and semantics explicit and composable. The implementation is object-oriented in Python and utilizes a small set of classes. A logical key identifies each cluster entity by combining its API group and version, kind, name, and, when present, namespace and subresource. Nodes represent resources and carry the local information needed for reasoning, in particular the set of admissible verbs and the effective labels. Pod nodes extend this representation with an indicator that records selection by a NetworkPolicy, a concept which is detailed subsequently. A set of typed `Edge` objects is used exclusively to encode the `CONTAINS` relation of the structural backbone. In contrast, the policy overlays are represented by dedicated data structures:

RBAC relationships are stored in specialized lists and dictionaries, while network flows are captured in separate maps for ingress and egress traffic. Finally, the *Graph* structure gathers the set of nodes, the containment edges, and these specialized overlay structures.

The resource backbone is constructed from a collection of input resources using a typed *containment* relationship. A root node, labeled `cluster`, is initially created. Beneath this root, a *Namespace bucket* is established to function as the logical container for all namespaces; every discovered `Namespace` instance is then connected to this bucket using the containment relation. For resources identified as cluster scoped kinds, each instance is attached directly beneath the root node. In contrast, for namespaced kinds, each namespace node serves as the anchor for one kind bucket per kind, and the concrete resource instances are consequently placed under the relevant bucket. Similarly, any subresources are placed beneath their parent instance. Through this process, the concept of a *bucket* is formalized: it exists as a class level node that represents all instances of a specific kind within a given scope, and simultaneously provides a canonical attachment point for defining policies that apply to the entire class without requiring the enumeration of every individual member. This entire methodology consistently mirrors the architectural design scheme detailed in Chapter 5, thereby preserving the same concepts of a resource backbone, namespace and kind buckets, and the fundamental containment relation.

The authorization model is built in a three-phase process. First, the links between **subjects**, such as users or service accounts, and **roles** are established. These associations are not added as simple edges to the graph. Instead, they are stored in a dedicated list structure, which conceptually represents all *bind* relations in the system. A key mechanism here is the creation of temporary, namespaced copies of cluster-wide roles, ensuring their powerful permissions are correctly scoped when assigned within a specific namespace. Next, the permissions for each role are resolved. The rules are translated into a specialized map that represents the *grant* relation. This structure efficiently connects each role to the resources it can access and the set of actions, or verbs, it is permitted to perform on them. This includes permissions granted to resource classes, known as *buckets*, as well as to specific instances. Finally, role aggregation is handled. For composite roles that inherit permissions from others, the system merges the grants from the member roles directly into the aggregator's entry in the permissions map. This process creates a complete and explicit view of its effective rights.

A similar approach is used for the network connectivity overlay. Instead of creating a generic “connects-to” edge for every allowed communication, the model uses a more nuanced representation. Nodes representing Pods are first flagged to indicate whether they are governed by ingress or egress policies. This is essential for correctly interpreting connectivity rules, distinguishing between a default-deny context where a policy applies, and a default-allow context where no policy applies. The allowed network flows are then captured in two distinct, directional maps.

- An **ingress map** associates each destination Pod with a list of its permitted sources and the corresponding port and protocol rules.
- An **egress map** associates each source Pod with its permitted destinations and rules.

It is worth noting that, in this iteration, the normalization process only applies the semantics of a supported subset of selectors; features such as `namespaceSelector` and `matchExpressions` are currently ignored to align expectations with the proof-of-concept’s behavior. This design makes the direction of traffic implicit in the choice of data structure, cleanly separating inbound from outbound policy and mirroring how Kubernetes itself defines them.

This implementation strategy fundamentally adheres to a core architectural principle: the clean separation between a stable **resource ontology** and the dynamic **logical policy rules**. The graph’s structural backbone defines the “what” and “where” (the resources and their hierarchy), while the specialized overlays determine the “who” and “how” (authority and connectivity). By representing conceptual relationships (such as *binding*, *granting*, and *network connection*) using dedicated data structures external to the main graph, the model preserves the graph’s clarity, keeping its focus strictly on containment. This separation significantly streamlines formal reasoning and facilitates the execution of powerful queries that span multiple policy domains.

#### 7.1.4 Translating the Graph into SWI-Prolog

To enable formal analysis, the object-oriented graph representation constructed in Python is translated into a queryable *Knowledge Base* (KB) in Prolog. This process relies on transforming the entire model into a set of **Prolog facts**. A fact is a declarative statement, a predicate asserted to be unconditionally true, which serves as a foundational atom of knowledge. For example, the statement “a Pod is contained in a Namespace” is converted from a relationship between Python objects into a simple, logical fact within the KB.

The translation is a systematic process of **factual serialization**, where the entities of the graph and their complex relationships are deconstructed into these atomic facts. This reshapes the structural and procedural information of the graph into a declarative format, making it amenable to the logical inference engine. The translation methodically processes the three main layers of the graph: the structural backbone, the RBAC overlay, and the NetworkPolicy overlay; generating a comprehensive set of facts that collectively represent the state of the Kubernetes cluster configuration.

The following table provides a detailed summary of each predicate generated during this translation process, outlining its structure and semantic role within the Knowledge Base.

longtable array booktabs

Predicate	Description
<i>Structural Backbone</i>	
node_full/3	Represents a single resource node in the graph. It asserts the node's unique key, its Kubernetes labels, and the set of admissible verbs. <i>Structure:</i> <code>node_full(key(...), Labels, Verbs)</code>
edge/2	Defines a direct CONTAINS relationship between two resource nodes, materializing the structural hierarchy. <i>Structure:</i> <code>edge(SourceKey, DestinationKey)</code>
<i>RBAC Overlay</i>	
bind/2	Represents an RBAC binding between a subject (User, Group, or ServiceAccount) and a Role or ClusterRole. <i>Structure:</i> <code>bind(Subject, Role)</code>
grant/3	Represents a specific permission. A separate fact is created for each verb granted by a role to a target. The target can be a specific resource ( <code>res(...)</code> ) or a class-level bucket ( <code>bucket(...)</code> ). <i>Structure:</i> <code>grant(Role, Target, Verb)</code>
aggregation/2	Defines an aggregation rule, where one ClusterRole inherits all permissions from another. <i>Structure:</i> <code>aggregation(AggregatorRole, MemberRole)</code>
clusterrole_copy_of/3	Records the creation of a virtual, namespaced copy of a ClusterRole that results from a RoleBinding. <i>Structure:</i> <code>clusterrole_copy_of(OriginalCR, Namespace, CopiedRole)</code>
<i>NetworkPolicy Overlay</i>	
np_flag/2	Indicates that a Pod is selected by a NetworkPolicy for a specific traffic direction. <i>Structure:</i> <code>np_flag(pod(...), Direction)</code>
np_ingress/5	Defines an allowed ingress traffic rule, specifying the destination pod, the source peer, connection details, and the originating policy. <i>Structure:</i> <code>np_ingress(DstPod, SrcPeer, Proto, Port, Policy)</code>

Predicate	Description
<code>np_egress/5</code>	<p>Defines an allowed egress traffic rule, specifying the source pod, the destination peer, connection details, and the originating policy.</p> <p><i>Structure:</i> <code>np_egress(SrcPod, DstPeer, Proto, Port, Policy)</code></p>

### 7.1.5 The Analysis Logic: Prolog Rules and Queries

Once the configuration graph is translated into a factual KB, the system requires a logical engine to interpret and analyze these facts. In the Proof of Concept, this logic is implemented through a set of Prolog rules and a direct query mechanism.

It is important to note that the PoC does not implement a parser for the high-level query language defined; the queries (called “goals” in Prolog) are formulated directly in Prolog syntax within the Python code. This approach allows for the full expressiveness of the logical language without the overhead of an intermediate translation layer. A query is sent to the Prolog engine, which attempts to satisfy it (i.e., find a solution) based not only on the existing facts but also on a set of predefined **logical rules**.

These rules constitute the true “intelligence” of the analysis system, as they define high-level concepts and complex relationships from atomic facts. For example, instead of manually checking a long chain of facts to determine a permission, it is possible to query a single rule that encapsulates the entire logic. The implemented rules can be conceptually grouped:

- **Verification Rules:** Predicates that answer direct questions with “yes” or “no,” such as `subject_has_role/3`, which checks if a subject is bound to a role.
- **Inference Rules:** More complex predicates that deduce new knowledge. The most significant example is `subject_has_perm/6`, which combines the logic of bindings with that of permissions to determine if a subject has an effective authorization on a resource.

The inference logic of `subject_has_perm` is powerful because its verification strategy adapts to the query’s specificity, directly reflecting the *Grant* semantics defined in the architecture.

When a query targets a resource identified by a **specific name**, the governing rule is satisfied if the subject possesses a permission granted either directly on that individual resource instance or on the encompassing *resource class* (or **bucket**) to which the instance belongs. This latter condition utilizes the architectural principle of **subsumption**, meaning that permissions defined for a resource bucket are automatically inherited by all contained instances.

Conversely, if the query omits the resource name, it fundamentally seeks permission across the entire resource class within the specified namespace. The

rule primarily verifies this by searching for a single, efficiently checked permission granted directly on the resource bucket itself. As an alternative, the rule can also succeed by confirming that the subject holds explicit permissions on *every single* concrete instance belonging to that resource kind. This two pronged strategy allows a single predicate to accurately manage both fine grained, instance level permission checks and broader, class level authorizations, aligning with the model’s design to target either specific resources or full resource sets.

A key aspect of formulating queries is handling **wildcards**. To represent concepts such as “any resource” or “at least one permission,” which in an abstract notation might be expressed with operators like ‘\*’ or ‘+’, Prolog’s **anonymous variable**, represented by the underscore character (`_`), is used.

In Prolog, the anonymous variable acts as a placeholder for a value whose specific content is not of interest, only its existence. When `_` appears in a goal, the Prolog engine will try to unify it with any term that satisfies the rule, without returning its value. This perfectly matches the semantics of an existential wildcard. For example, to verify if a ServiceAccount `sa1` has the ‘get’ permission on *any* Pod in the `webapp` namespace, the query does not enumerate all Pods but uses the anonymous variable:

```
?- subject_has_perm(sa('webapp','sa1'), 'core/v1', 'Pod',
    'webapp', _, none, 'get').
```

In this query, the `_` symbol in place of the Pod’s name instructs the engine to search for *at least one* fact or chain of inferences that satisfies the rule for any Pod, making the query concise and efficient.

## 7.2 Validation

While the previous section described the implementation details of the Proof of Concept, this section demonstrates its practical application through experimental validation. The goal is to translate the architecture and the logical model into tangible, verifiable results, showing how the system responds to concrete queries in a realistic scenario.

To this end, the Kubernetes cluster used as the test environment will first be presented, describing its topology, resources, and security configurations to provide an operational context. Subsequently, a series of queries designed to test the system’s capabilities will be illustrated. These queries were chosen to cover significant use cases that require cross-domain analysis of RBAC permissions and network connectivity policies, thereby demonstrating the expressiveness of the language and the power of the inference engine. The analysis of the obtained results will serve to empirically validate the proposed approach, confirming that the model is capable of correctly answering high-level questions through the definition of the created language and their translation into Prolog.

## 7.2.1 Cluster Description

For the validation of the Proof of Concept, a scenario simulating a web application in a Kubernetes cluster was set up. The area of interest is limited to the `webapp` namespace, which exclusively hosts the relevant application components and their custom policies. To maintain focus on the application perimeter, only the resources defined by a potential *developer* or *namespace admin* are presented; system resources and objects automatically created by the control plane are present in the snapshot, but are not described for the purposes of this demo.

**Application Components.** Two *Deployments* and their respective *Pods* have been configured in the `webapp` namespace:

- **Frontend:** Deployment/frontend with Pod frontend-58b4d497c-5d19c (app=frontend).
- **Backend:** Deployment/backend with Pod backend-54f4d65bb7-dps7q (app=backend).

**Application Identities.** Two *ServiceAccounts* (`sa1`, `sa2`) are defined in the `webapp` namespace; additionally, an application user `user1` is used for RBAC verifications. Any default accounts or auxiliary objects are not covered in this description.

**RBAC Roles and Permission Semantics.** The permission configuration has been set up following the principle of least privilege, distinguishing between namespaced roles and cluster roles:

- **Role pod-reader (@webapp):** allows only the `get` verb on pods within the `webapp` namespace.
- **Role frontend-developer (@webapp):** grants `get`, `watch`, `list`, `create`, `update` on pods, limiting its effectiveness to the single instance frontend-58b4d497c-5d19c via `resourceNames`.
- **ClusterRole base-get-permissions:** enables `get` on all resources in the core group; it is labeled for aggregation.
- **ClusterRole deployment-creator-aggr:** enables `create` and `update` on deployments; it is labeled for aggregation.
- **ClusterRole get-reader-aggr (aggregator):** declares no rules of its own; through an `aggregationRule`, it automatically includes labeled `ClusterRoles`, resulting in a profile that combines `get` on core resources and `create/update` on deployments.

**Bindings between Subjects and Roles.** The following *RoleBindings* are defined in the `webapp` namespace:

- `pod-reader-binding`: associates `sa1` with the Role `pod-reader`.
- `frontend-dev-binding`: associates `sa2` with the `frontend-developer` Role.
- `get-reader-binding-webapp-user1`: associates `user1` with a `ClusterRole` via a namespaced *RoleBinding*. As explained in Section 7.1.3 on implementation, in this configuration, the PoC’s logic introduces a *namespaced copy* of the `ClusterRole` so that its permissions are effective exclusively within `webapp`. Since `get-reader-aggr` aggregates `base-get-permissions` and `deployment-creator-aggr`, the effective profile of `user1` in the namespace includes `get` on core resources and `create/update` on deployments, confined to the namespace perimeter.

**Network Policies.** Connectivity is governed by two *NetworkPolicies* in the `webapp` namespace:

- `allow-backend-from-frontend` (*Ingress*): selects Pods with `app=backend` and allows incoming TCP/5000 traffic from Pods with `app=frontend`.
- `allow-frontend-egress` (*Egress*): selects Pods with `app=frontend` and allows outgoing traffic to `0.0.0.0/0` exclusively on TCP/80 and TCP/443.

**Note on the Presented Perimeter.** System resources and automatically generated objects (e.g., default accounts and roles, internal controllers, auxiliary resources) are present in the snapshot and contribute to the fidelity of the context, but they are not discussed in the text as they are not necessary for understanding the use cases and are not attributable to explicit choices by the application team.

## 7.2.2 Query Examples and Results

To demonstrate the capabilities of the Proof of Concept, a series of queries were implemented and tested, covering various aspects of security in the `webapp` cluster, including RBAC authorizations and network connectivity rules. Each example includes the syntax in the high-level language, its discursive meaning, and the translation into the corresponding Prolog query.

## Binding Verification Queries

The following queries focus on the direct verification of associations (binding) between subjects and roles, without inferring the effective permissions.

### Query 1: Who is bound to what? (Specific Role)

This fundamental query is formulated to check for the existence of a direct association (a “binding”) between the subject `user1` and the role `get-reader`

within the specific `webapp` namespace. It does not query the effective permissions derived from the role, only the presence of the binding declaration.

- **Query:**

```
[user(user1)] has( [Role(get-reader@webapp)] )
```

- **Query in Prolog:**

```
?- subject_has_role(user('user1'), 'get-reader',
    'webapp').
```

- **Result: TRUE.**

The result is affirmative because, as described in the “Bindings between Subjects and Roles” section, a `RoleBinding` named `get-reader-binding-webapp-user1` is present. This object explicitly declares that the subject `user1` is associated with the `ClusterRole` named `get-reader` within the `webapp` namespace. The `subject_has_role` query is designed to find exactly this type of declared association, so the verification succeeds.

## Query 2: Who is bound to what? (Generic Role)

Expanding on the previous example, this more generic inquiry asks whether the user `user1` has an association with *any role* (it does not matter which one) within the `webapp` namespace. The objective is to verify the existence of at least one binding for that user in that specific context.

- **Query:**

```
[user(user1)] has( +[Role(*@webapp)] )
```

- **Query in Prolog:**

```
?- subject_has_role(user('user1'), _NS, 'webapp').
```

As anticipated in the section on the implementation of the analysis logic, a variable starting with an underscore (`_NS` in this case) is an **anonymous variable** and acts as a wildcard. Its use indicates to the Prolog engine that we are not interested in the specific value of the role’s name, but only in the existence of at least one association that satisfies the other known parameters (the subject `user1` and the namespace `webapp`). If Prolog finds any value for `_NS` that makes the statement true, the query succeeds.

- **Result: TRUE.**

The result is affirmative. The Prolog engine searches its facts for an association matching `subject_has_role(user('user1'), _Anything, 'webapp')`. It finds the fact generated by the `RoleBinding` `get-reader-binding-webapp-user1`, which links `user1` to the `get-reader` role in the `webapp` namespace. The anonymous variable `_NS` successfully unifies with the role name `'get-reader'`, satisfying the query.

### Query 3: Who is bound to what? (Reverse Lookup)

Inverting the logic of the preceding examples, this query starts from a specific role (`pod-reader`) and asks if a specific subject (`sa2`) is associated with it.

- **Query:**

```
[role(pod-reader@webapp)] has( [sa(sa2@webapp)])
```

- **Query in Prolog:**

```
?- role_has_subject(role(role, 'pod-reader', 'webapp'),
  sa('webapp', 'sa2')).
```

- **Output: FALSE.**

The result is negative. Analyzing the cluster configuration, the `pod-reader` role is exclusively associated with the `ServiceAccount sa1` via the `pod-reader-binding`. The `ServiceAccount sa2`, on the other hand, is associated with a completely different role, `frontend-developer`. Since no binding declaration exists that links the `pod-reader` role to the `sa2` subject, the query correctly fails.

### Query 4: Explore & List (Subject's Roles)

This exploratory query moves beyond a simple boolean check, instructing the system to enumerate all roles to which a given subject is associated. Specifically, it asks to identify which roles are associated with the `ServiceAccount sa1` within the `webapp` namespace.

- **Query:**

```
[sa(sa1@webapp)] list( Roles ,
  has( +[Permission( [*] , [*] ) ] ) )
```

- **Query in Prolog:**

```
?- subject_roles(sa('webapp', 'sa1'), Roles).
```

- **Output:** `role(role, 'pod-reader', webapp).`

The result clearly indicates that the `ServiceAccount sa1` is exclusively linked to the `pod-reader` role within the `webapp` namespace. This finding perfectly aligns with the provided configuration, given that the `pod-reader-binding` is the single binding defined for `sa1`.

The `subject_roles` query is specifically engineered to aggregate all roles that are directly associated with a designated subject, thereby generating an explicit and complete list of its declared policy connections.

## Permission Verification Queries

### Query 5: Can I? (Specific Permission on Generic Resource)

This inquiry shifts the focus from role definitions to effective permissions for a specific subject. It is designed to determine if the `ServiceAccount sa1` in the `webapp` namespace has the authorization to perform the `get` action on *any* resource of type `Pod` within the same namespace.

- **Syntax in the Defined Language:**

```
[sa(sa1@webapp)] has( [Permission( [get],
    [Pod@webapp#*])])
```

- **Query in Prolog:**

```
?- subject_has_perm(sa('webapp', 'sa1'), 'core/v1', 'Pod',
    'webapp', _Name, none, 'get').
```

- **Output:** `TRUE.`

The `ServiceAccount sa1` is associated with the `pod-reader` role via the `pod-reader-binding`. As described, the `pod-reader` role explicitly grants the `get` permission on all `Pods` resources in the `webapp` namespace. The Prolog query uses the anonymous variable `_Name` to indicate that the specific name of the `Pod` is not relevant, looking for a valid permission for any `Pod`. The system's inference identifies this permission and responds affirmatively.

### Query 6: Can I? (Generic Permission on Any Resource)

This is a broad, exploratory permission check. The intent is to determine if `ServiceAccount sa1` possesses a universal read permission, specifically by

asking if it can perform the `get` action on *any type of resource* in *any API Group* in the cluster.

- **Query:**

```
[sa(sa1@webapp)] has( +[Permission( [get], [*])])
```

- **Query in Prolog:**

```
?- subject_has_perm_piu(sa('webapp','sa1'), _ApiGV,
    _Kind,
    _Ns, _Name, _Sub, 'get').
```

- **Output: TRUE.**

Although the `pod-reader` role is specific to Pods, the system checks for the existence of *at least one* `get` permission on *any resource*. Since `sa1` has `get` on Pods, and Pods are a type of resource, the “any resource” condition is met. The query uses several anonymous variables (`_ApiGV`, `_Kind`, etc.) to indicate that the system must find an instance of a `get` permission that meets the criteria without specifying the API Group, Kind, Namespace, name, or subresource. Given the presence of the `get` permission on Pods, the system finds a match.

## Connectivity Verification Queries

The following queries focus on validating the network rules defined by the NetworkPolicies.

### Query 7: Can it talk? (Ingress Reachability)

This query probes the network policy layer to validate a specific connectivity rule. Its purpose is to confirm the existence of a fact in the model that represents an allowed *ingress* traffic flow, where the `backend` Pod is the destination, the `frontend` Pod is the source, and the communication occurs on port TCP/5000.

- **Query:**

```
[NE(Pod@webapp#frontend-58b4d497c-5d19c)] has (
    [Connection(
        Pod@webapp#backend-54f4d65bb7-dps7q, 5000, TCP,
        ingress )] )
```

- **Query in Prolog:**

```
?- np_ingress(dst_pod('webapp',
    'backend-54f4d65bb7-dps7q'),
    src(pod('webapp', 'frontend-58b4d497c-5d19c')),
    proto('TCP'), port('5000'), policy(_P)).
```

- **Output: TRUE.**

The `NetworkPolicy` named `allow-backend-from-frontend` defines this exact rule: it selects Pods with the label `app=backend` and allows incoming traffic on port `TCP/5000` from Pods with the label `app=frontend`. During the process of translating the cluster into a Prolog knowledge base, this rule is converted into an `np_ingress` fact with the exact parameters specified in the query. Since the query asks for the existence of this specific fact, and the fact is present, the verification succeeds.

## Cardinality Queries

### Query 8: Authorization Cardinality (Roles)

This query tests not just for existence, but for quantity against a threshold. It asks whether the `ServiceAccount sa1` is associated with “more than one” role, checking for potentially excessive permissions or misconfigurations.

- **Query:**

```
[sa(sa1@webapp)] list( Roles, has( +[ Role(*) ])) > 1
```

- **Query in Prolog:**

```
?- subject_roles_rel(sa('webapp', 'sa1'), gt, 1).
```

The predicate `subject_roles_rel/3` is designed to count the roles associated with a subject and compare the count with a given value, using a relational operator (here `gt` for “greater than”).

- **Output: FALSE.**

The result is negative. As seen in previous queries, the `ServiceAccount sa1` is solely associated with the `pod-reader` role. Therefore, `sa1` has exactly one role, and the condition “more than one” (`> 1`) is not met. The query demonstrates the system’s ability to perform aggregations and comparisons on the data, in addition to simple fact lookups.

## Cross-Domain Queries (RBAC and Network Policy)

The following example demonstrates the system’s ability to answer complex questions that span multiple policy domains, combining access and network constraints into a single query.

### Query 9: Dangerous Combinations

This represents a critical and complex security audit. The query is constructed to identify if two potentially dangerous conditions exist concurrently: 1) that the `ServiceAccount sa1` has permission to execute commands inside a pod (the `create` verb on the `exec` subresource), AND 2) that there is at least one pod in the same namespace that is allowed to initiate a connection to the outside world (egress to `0.0.0.0/0`).

- **Query:**

```
[sa(sa1@webapp)] has( [Permission( [create] ,
  [Pod/exec@webapp#*])] ) AND
+[NE(Pod@webapp#*)] has( +[Connection(
  0.0.0.0/0, *, *, egress)] )
```

- **Query in Prolog:**

```
?- subject_has_perm(sa('webapp', 'sa1'), 'core/v1', 'Pod'
  , 'webapp', _Any, 'exec', 'create')
  ,
  np_egress(src_pod('webapp', _P), dst(cidr('0.0.0.0/0')),
  proto(_), port(_), policy(_)).
```

In Prolog, the comma `,` between two goals acts as a logical **AND** operator. Therefore, the query succeeds only if *both* conditions (the one on permissions and the one on connectivity) are met.

- **Output: FALSE.**

The query fails because, although one of the two conditions is true, they are not both true. Analyzing the two parts separately:

- **RBAC Part** (`subject_has_perm(...)`): This condition is **FALSE**. The `ServiceAccount sa1` is exclusively associated with the `pod-reader` role, which only grants the `get` permission on Pods. It absolutely does not have the authorization to perform a critical operation like `exec`.
- **Network Policy Part** (`np_egress(...)`): This condition is **TRUE**. As defined in the cluster configuration, the

`NetworkPolicy allow-frontend-egress` explicitly allows the frontend Pod to establish outgoing connections to external addresses.

Since the AND operator requires both conditions to be true, and the RBAC condition is false, the entire query correctly fails. This demonstrates the system's ability to provide precise answers to questions that correlate different security policies.

## Manual Effort for Cross-Domain Verification

To fully appreciate the value of a cross-domain query like the previous one, it is essential to consider the effort a system administrator or operator would have to undertake to verify it manually. The manual process would require the following steps, each susceptible to errors or omissions:

(a) **RBAC Permission Check (Part 1: Finding Bindings):**

- The operator would have to identify the `ServiceAccount sa1` in the `webapp` namespace.
- Next, they would have to search for all `RoleBindings` and `ClusterRoleBindings` involving `sa1` in the `webapp` namespace.
- For each binding found, they would have to identify the associated `Role` or `ClusterRole`.

(b) **RBAC Permission Check (Part 2: Analyzing Roles):**

- For each role identified in the previous step, the operator would have to examine the rules to determine the effective permissions.
- Specifically, they would have to check if the `create` verb on the `Pod/exec` resource is granted. This requires detailed knowledge of Kubernetes subresource semantics and verb mappings.
- In the case of aggregated `ClusterRoles`, the operator would have to manually trace all `ClusterRoles` that contribute to the aggregation, expand their permissions, and check if the verb `create` on `pods/exec` is included.
- Finally, they would have to deduce whether `sa1` has this permission.

(c) **Network Connectivity Check (Part 1: Identifying Pods):**

- The operator would have to identify all Pods in the `webapp` namespace that are potential sources of outgoing traffic.
- For each of these Pods, they would have to examine its labels to understand which `NetworkPolicies` select it.

(d) **Network Connectivity Check (Part 2: Analyzing NetworkPolicies):**

- For each Pod selected by one or more `NetworkPolicies`, the operator would have to analyze the `egress` rules of each policy.
- They would have to check if any of these rules allow traffic to the destination `0.0.0.0/0` (any external) on any port or protocol.

- If there are no `egress NetworkPolicies` selecting a given Pod, or if no `egress` rule explicitly allows it, and the Pod is in an environment with a default-deny policy (a common practice), the traffic would be blocked. This requires an understanding of the default logic of `NetworkPolicies`.

(e) **Final Correlation:**

- Only after completing all independent checks would the operator have to mentally or manually combine the results of the two parts (RBAC and Network Policy) for the "AND" condition to answer the original question.

This manual process is extremely laborious, prone to human error (especially in complex configurations or with a large number of resources), requires deep knowledge of the internal details of Kubernetes and its APIs, and is not scalable. The proposed system automates the entire workflow, providing a deterministic and justifiable answer in a fraction of the time of its manual equivalent, drastically reducing the cognitive load on the operator.

### 7.2.3 Evaluation Time and Bottlenecks

The Proof of Concept exhibits low and stable latencies. The analysis is based on a snapshot of the PoC's demo environment (named `webapp` snapshot), which consisted of 521 nodes and 520 edges in the dependency graph, including 3 RBAC bindings, 4262 grants, and 3 Network Policy rules in total (1 ingress, 2 egress).

Graph construction from this cluster snapshot completes in approximately 272.66ms (inputs reading 0.67ms, backbone and bucket materialization 14.97ms, RBAC integration 252.62ms, NetworkPolicy integration 4.40ms). The dominant cost is the one-time serialization of the graph into the Prolog Knowledge Base through fact assertion, which requires about 5894.07ms.

Once the Knowledge Base (KB) is populated, goal evaluation proceeds within millisecond budgets. Table 7.3 reports the build-time breakdown. Table 7.4 summarizes indicative statistics for Prolog goal evaluation under warm conditions, derived from the implemented queries executed against the fixed KB representing the PoC snapshot. These queries were categorized as follows:

- **RBAC predicate (verification):** Queries checking a specific authorization path (e.g., *can user X perform action Y on resource Z?*).
- **RBAC function query (listing/counting):** Queries listing or counting related entities (e.g., *which ServiceAccounts are bound to ClusterRole Z?*).
- **NetworkPolicy predicate (reachability):** Queries verifying network reachability between specific Pods (e.g., *can Pod A reach Pod B on port P?*).

- **Cross-domain (RBAC & NP, single goal):** A query combining both authorization and connectivity in a single logical goal (e.g., *can ServiceAccount X successfully perform an action that requires network connection to Pod B?*). This measures the true cost of combined verification.

These figures are intended primarily to illustrate the relative processing weights within the Proof of Concept (PoC) dataset and its inherent simplifications. In a complete, production grade implementation, the absolute runtimes will certainly increase, but the asymmetric cost profile will remain constant. The most resource intensive steps are the initial, one time construction of the typed graph from the cluster snapshot and the subsequent translation of this graph into the Prolog knowledge base. Once this foundational state is fully constructed and loaded, subsequent administrator queries are evaluated directly against the fixed knowledge base. Consequently, each query only incurs the minimal reasoning cost, which consistently falls within the sub millisecond to low millisecond range. The initial setup latency is thus fully amortized across many subsequent queries, and this cost is only incurred again when the cluster snapshot is refreshed and the graph and knowledge base require rebuilding.

Table 7.3. Indicative PoC timings and sizes (single run, `webapp` snapshot)

Stage (graph build)	Time (ms)
Read inputs (resources, RBAC, NP)	0.67
Backbone kind buckets build	14.97
RBAC integration (binds, grants, aggregation)	252.62
NetworkPolicy normalization	4.40
<b>Total (graph only)</b>	<b>272.66</b>
<b>Graph → Prolog KB (assert facts)</b>	<b>5894.07</b>

Table 7.4. Indicative Prolog goal evaluation times (mean over warm runs; PoC snapshot)

Query category	Mean (ms)
RBAC predicate (verification)	0.65
RBAC function query (listing/counting)	0.59
NetworkPolicy predicate (reachability)	0.67
Cross-domain (RBAC & NP, single goal)	1.04

# Chapter 8

## Conclusions

This thesis successfully addresses the critical challenge of verifying complex, fragmented security policies in Kubernetes. The opaque and non-deterministic nature of combining policies across distinct domains, specifically **authorization (RBAC)** and **network connectivity (NetworkPolicy)**, forces operators into manual, error-prone verification. Our work resolves this complexity by delivering a systematic and formally rigorous approach that transitions security verification from reactive inspection to proactive, engineering-grade assurance.

Our work achieved four fundamental objectives that together form a comprehensive verification framework. First, we enabled **Cross-Domain Reasoning** by unifying inherently heterogeneous policy sources (RBAC and NetworkPolicy) into a single, coherent typed graph model, thereby eliminating data silos and allowing for rigorous analysis of combined policy effects. Building on this, we met the objective of **Grounding a Disciplined Model** by establishing this representation within a typed, extensible foundation that can evolve to accommodate new policy families without compromising internal consistency. We then ensured **Deterministic and Explicit Reasoning** by maintaining a strict focus on semantic explicitness, guaranteeing that the reasoning engine produces reproducible and explainable results regardless of policy changes. Finally, by designing and implementing a dedicated query language that balances formal soundness with operational readability, we succeeded in **Exposing Verification Capabilities**, enabling operators to execute sophisticated, non-trivial verification tasks with precision. In summary, this research provides the necessary foundation for automating the verification of Kubernetes security posture.

The proposed architecture meets these goals by unifying policy artifacts into a single typed graph, which is then compiled into a **Prolog knowledge base** for logic-based reasoning. The model accurately captures authorization (RBAC bindings, grants) and connectivity (NetworkPolicy selection and directional rules). Queries are expressed in a purpose-built language that is readable, formally grounded, and compositional, covering boolean checks, value computations, and counting constraints. This emphasis on clarity and

deterministic evaluation ensures consistency: the same configuration yields the same answer, and the query semantics are stable across scenarios.

The proof of concept (PoC) confirms the feasibility and practical value of the approach. Using a realistic snapshot, we demonstrated that the computationally heavy work of graph construction and compilation is performed once per snapshot ( $\sim 6000$  ms total), and the resulting state is then reused. Subsequent query evaluation remains in the sub-millisecond to low-millisecond range. This separation of one-time preparation and repeated, low-latency evaluation is essential for continuous assurance workflows.

The query language itself is a central contribution. Its type system prevents invalid combinations, guiding users to construct meaningful, auditable verification expressions. This language serves as the public interface of the methodology, allowing precise, repeatable cross-domain verification without exposing low-level model details.

The scope of the results is bounded by two factors: the PoC validates the architectural core exclusively on RBAC and NetworkPolicy, and the evaluation considers only a subset of Kubernetes API groups/resources rather than full API coverage. These constraints were chosen for clarity and determinism; the positive evidence motivates extending both policy domains and API surface in future work.

## 8.1 Future Work

The results achieved open several clear paths for future development, beginning with the push for production-readiness and scalability. The immediate priority is a complete, robust implementation, which will require building reliable ingestion pipelines to acquire configuration snapshots from live clusters and **Infrastructure-as-Code (IaC)** repositories. To manage performance at scale, continuous reconciliation must be implemented using mechanisms that propagate deltas to the graph and knowledge base instead of triggering costly full rebuilds. Furthermore, the query language needs to be exposed through secure, stable interfaces for seamless integration into **CI/CD** pipelines and operational tooling. For large-scale environments, partitioning the graph and knowledge base will be necessary to support parallel construction and distributed evaluation.

Beyond implementation, the formal model must be expanded to cover more security dimensions. This involves integrating **Attribute-Based Access Control (ABAC)** to support context-aware authorization decisions based on subject or environment attributes. A crucial expansion is linking workload identity and namespaces to data handling paths by explicitly modeling pod-security constraints and memory/volume access. Volume access, including both ephemeral and persistent volumes managed through **CSI**, must be carefully reconciled with storage backends. At the application layer, integrating **service-mesh** and Layer-7 policies (e.g., Istio) with the existing

transport-layer connectivity view will provide continuity from Layer 3 to application flows. Finally, for hybrid environments, defining explicit mappings between Kubernetes identities and cloud **IAM** roles is essential to enable cross-boundary verification.

Advanced reasoning should focus on enabling proactive simulation capabilities. This means implementing “**what-if**” queries, or **Counterfactual Analysis**, that simulate the impact of proposed policy changes (e.g., removing a binding or tightening a network rule) before deployment, thereby transforming the system into a policy simulator. Extending the query language to naturally map representative queries to industry compliance controls will also enable continuous auditing and reporting.

## Closing Remarks

This work demonstrates that a unified, logic-based approach can successfully transform policy verification in Kubernetes into a coherent, deterministic workflow. By providing a formally sound model, a precise language, and a validated pipeline, the thesis proves that complex, cross-domain security questions can be answered systematically. The architectural principles are highly generalizable, providing a clear and disciplined path to incorporate future policy families while maintaining semantic clarity and deterministic results, establishing a practical foundation for continuous assurance in cloud-native security.

# Bibliography

- [1] National Security Agency and Cybersecurity & Infrastructure Security Agency, “Kubernetes hardening guidance,” Tech. Rep. v1.2, Aug 2022, recommendations to avoid common misconfigurations; emphasizes RBAC least privilege and network separation. [Online]. Available: [https://media.defense.gov/2022/Aug/29/2003066362/-1/-1/0/CTR\\_KUBERNETES\\_HARDENING\\_GUIDANCE.1.2.20220829.PDF](https://media.defense.gov/2022/Aug/29/2003066362/-1/-1/0/CTR_KUBERNETES_HARDENING_GUIDANCE.1.2.20220829.PDF)
- [2] Kubernetes Authors. (2025, Apr) Using rbac authorization. Official documentation for Kubernetes RBAC. [Online]. Available: <https://kubernetes.io/docs/reference/access-authn-authz/rbac/>
- [3] ——. (2024, Apr) Network policies. Official documentation for Kubernetes NetworkPolicies. [Online]. Available: <https://kubernetes.io/docs/concepts/services-networking/network-policies/>
- [4] OWASP Foundation. (2022) Owasp kubernetes top ten (2022). K03 Overly Permissive RBAC, K04 Lack of Centralized Policy Enforcement, K07 Missing Network Segmentation Controls. [Online]. Available: <https://owasp.org/www-project-kubernetes-top-ten/>
- [5] MITRE ATT&CK. (2021, Mar) Mitre att&ck for containers: Deploy container (t1610) – mitigations. Highlights Network Segmentation and account controls as mitigations. [Online]. Available: <https://attack.mitre.org/techniques/T1610/>
- [6] A. Rahman, S. I. Shamim, D. B. Bose, and R. Pandita, “Security misconfigurations in open source kubernetes manifests: An empirical study,” *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 4, pp. 99:1–99:36, 2023. [Online]. Available: <https://doi.org/10.1145/3579639>
- [7] A. Sissodiya, E. Chiquito, U. Bodin, and J. Kristiansson, “Formal verification for preventing misconfigured access policies in kubernetes clusters,” *IEEE Access*, vol. 13, pp. 141 798–141 813, 2025. [Online]. Available: <https://doi.org/10.1109/ACCESS.2025.3597504>
- [8] F. Valenza, M. Cheminod, and A. Liroy, “An optimized approach for assisted firewall anomaly resolution,” *Journal of Internet Services and Information Security*, vol. 10, no. 1, pp. 22–37, 2020. [Online]. Available: <https://doi.org/10.22667/JISIS.2020.02.29.022>
- [9] D. Bringhenti, G. Marchetto, R. Sisto, and F. Valenza, “Automation for network security configuration: State of the art and research trends,” *ACM Computing Surveys*, vol. 56, no. 3, pp. 57:1–57:37, 2024. [Online]. Available: <https://doi.org/10.1145/3616401>

- [10] D. Brighenti, S. Bussa, R. Sisto, and F. Valenza, “Atomizing firewall policies for anomaly analysis and resolution,” *IEEE Transactions on Dependable and Secure Computing*, vol. 22, no. 3, pp. 2308–2325, 2025. [Online]. Available: <https://doi.org/10.1109/TDSC.2024.3495230>
- [11] A. Westerinen, J. Schnizlein, J. Strassner, M. Scherling, B. Quinn, S. Herzog, A. Huynh, M. Carlson, J. Perry, and S. Waldbusser, “Terminology for policy-based management,” *RFC*, vol. 3198, pp. 1–21, 2001. [Online]. Available: <https://doi.org/10.17487/RFC3198>
- [12] M. Cheminod, L. Durante, L. Seno, F. Valenza, and A. Valenzano, “A comprehensive approach to the automatic refinement and verification of access control policies,” *Computers & Security*, vol. 80, pp. 186–199, 2019. [Online]. Available: <https://doi.org/10.1016/j.cose.2018.09.013>
- [13] D. Brighenti, G. Marchetto, R. Sisto, S. Spinoso, F. Valenza, and J. Yusupov, “Improving the formal verification of reachability policies in virtualized networks,” *IEEE Transactions on Network and Service Management*, vol. 18, no. 3, pp. 713–728, 2021. [Online]. Available: <https://doi.org/10.1109/TNSM.2020.3045781>
- [14] D. Brighenti, S. Bussa, R. Sisto, and F. Valenza, “A two-fold traffic flow model for network security management,” *IEEE Transactions on Network and Service Management*, vol. 21, no. 4, pp. 3740–3758, 2024. [Online]. Available: <https://doi.org/10.1109/TNSM.2024.3407159>
- [15] F. Pizzato, D. Brighenti, R. Sisto, and F. Valenza, “An intent-based solution for network isolation in kubernetes,” in *2024 IEEE International Conference on Network Softwarization (NetSoft)*, 2024, pp. 381–386. [Online]. Available: <https://doi.org/10.1109/NETSOFT60951.2024.10588939>
- [16] —, “Security automation in next-generation networks and cloud environments,” in *2024 IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2024. [Online]. Available: <https://doi.org/10.1109/NOMS59830.2024.10575650>
- [17] Kubernetes Authors. (2024, Sep) Overview. Includes sections: “Why you need Kubernetes and what it can do”, “What Kubernetes is not”, and historical context. Last modified Sep 11, 2024. [Online]. Available: <https://kubernetes.io/docs/concepts/overview/>
- [18] —. (2025) Control plane–node communication. Guidance on secure communication between control plane and nodes. Accessed: 2025-09-04. [Online]. Available: <https://kubernetes.io/docs/concepts/architecture/control-plane-node-communication/>
- [19] —. (2025) Kubernetes components. Official documentation for control plane and node components. Accessed: 2025-09-04. [Online]. Available: <https://kubernetes.io/docs/concepts/overview/components/>
- [20] —. (2025) Kubernetes objects. API object model: kind, apiVersion, metadata, spec/status, subresources. Accessed: 2025-09-04. [Online]. Available: <https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/>
- [21] —. (2025) Labels and selectors. Equality and set-based selectors; distinction from annotations. Accessed: 2025-09-04. [Online]. Available: <https://kubernetes.io/docs/concepts/overview/>

- [working-with-objects/labels/](#)
- [22] ——. (2025) Namespaces. Virtual clusters within a physical cluster; scoping and isolation. Accessed: 2025-09-04. [Online]. Available: <https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/>
  - [23] ——. (2025) Pods. Pod model, probes, lifecycle, containers sharing namespaces. Accessed: 2025-09-04. [Online]. Available: <https://kubernetes.io/docs/concepts/workloads/pods/>
  - [24] ——. (2025) Deployments. ReplicaSets, rollouts/rollbacks, scaling, immutability of selectors. Accessed: 2025-09-04. [Online]. Available: <https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>
  - [25] ——. (2025) Statefulsets. Stable network IDs, ordered deployment, persistent storage. Accessed: 2025-09-04. [Online]. Available: <https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/>
  - [26] ——. (2025) Daemonsets. Node-wide workloads; automatic placement on new nodes. Accessed: 2025-09-04. [Online]. Available: <https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/>
  - [27] ——. (2025) Jobs. Run-to-completion workloads, retries, parallelism. Accessed: 2025-09-04. [Online]. Available: <https://kubernetes.io/docs/concepts/workloads/controllers/job/>
  - [28] ——. (2025) Cronjobs. Time-based scheduling for Jobs (cron syntax). Accessed: 2025-09-04. [Online]. Available: <https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/>
  - [29] ——. (2025) Service. Virtual IP/DNS, selectors, ClusterIP/NodePort/LoadBalancer. Accessed: 2025-09-04. [Online]. Available: <https://kubernetes.io/docs/concepts/services-networking/service/>
  - [30] ——. (2025) Configmaps. Externalized configuration; env vars and volumes. Accessed: 2025-09-04. [Online]. Available: <https://kubernetes.io/docs/concepts/configuration/configmap/>
  - [31] ——. (2025) Secrets. Sensitive data handling; encryption at rest recommendations. Accessed: 2025-09-04. [Online]. Available: <https://kubernetes.io/docs/concepts/configuration/secret/>
  - [32] ——. (2025) Persistent volumes. PVs, PVCs, StorageClass, access/volume modes, reclaim policy. Accessed: 2025-09-04. [Online]. Available: <https://kubernetes.io/docs/concepts/storage/persistent-volumes/>
  - [33] ——. (2023, Jun) Controlling access to the kubernetes api. Overview of API request flow: TLS, authentication, authorization, admission, validation. Accessed: 2025-09-04. [Online]. Available: <https://kubernetes.io/docs/concepts/security/controlling-access/>
  - [34] ——. (2025, Aug) Authentication. Authentication strategies (client certs, bearer/OIDC tokens, SA tokens, authenticating proxy); identity attributes and groups. Accessed: 2025-09-04. [Online]. Available: <https://kubernetes.io/docs/reference/access-authn-authz/authentication/>
  - [35] ——. (2024, Nov) Service accounts. ServiceAccount object model, token projection via TokenRequest API, audiences and rotation. Accessed: 2025-09-04. [Online]. Available: <https://kubernetes.io/docs/concepts/security/service-accounts/>

- [36] ——. (2025, Jun) Admission controllers. Mutating vs. validating admission controllers; request evaluation and enforcement before persistence. Accessed: 2025-09-04. [Online]. Available: <https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/>
- [37] A. K. Bandara, E. Lupu, J. D. Moffett, and A. Russo, “A goal-based approach to policy refinement,” in *5th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2004)*, 7-9 June 2004, Yorktown Heights, NY, USA, 2004, pp. 229–239. [Online]. Available: <https://doi.org/10.1109/POLICY.2004.1309175>